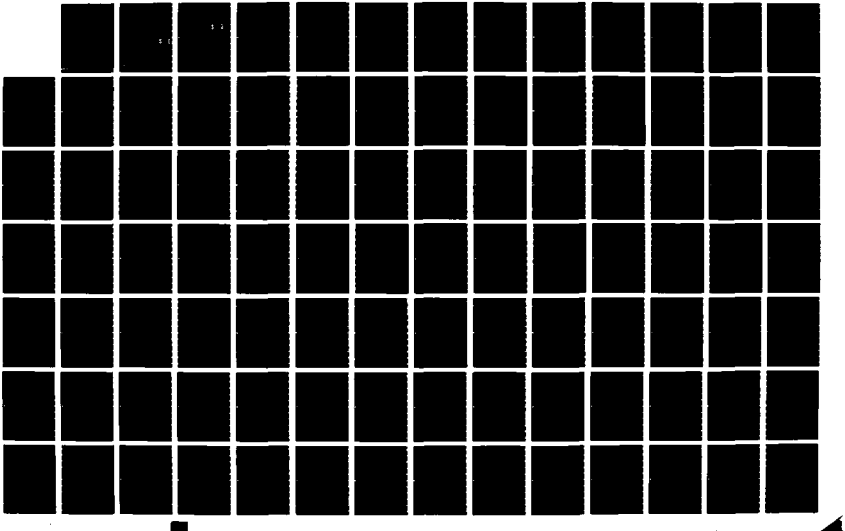
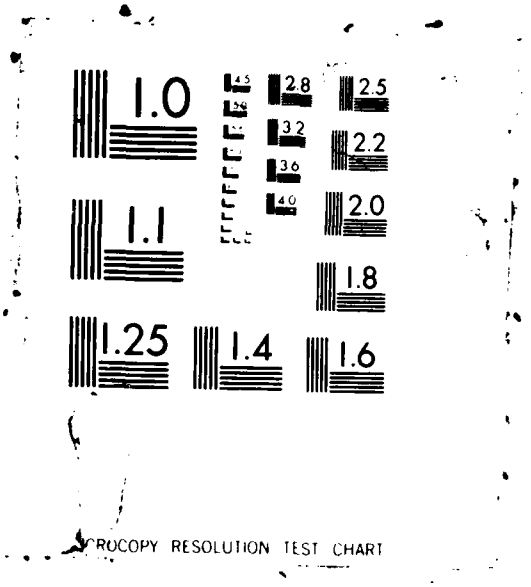
AD-A177 943 A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING C J TITUS DEC 86 AFIT/GE/ENG/86D-59 1/2

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
ROCOPY RESOLUTION TEST CHART

A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA

THESIS

Christopher J. Titus
Captain, USAF

AFIT/GE/ENG/86D-59

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/86D-59

A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA

THESIS

Christopher J. Titus
Captain, USAF

AFIT/GE/ENG/86D-59

AFIT/GE/ENG/86D-59

A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Christopher J. Titus, B.S.

Captain, USAF

December 1986

Approved for public release; distribution unlimited

Acknowledgements

A number of individuals deserve acknowledgement for their contributions and support during the course of this project. I would like to thank my thesis advisor, Dr Matthew Kabrisky, for his guidance and encouragement when my goals were unclear, and Major David King for his help with the computer system. Special recognition goes to my thesis sponsors, Mr Neal Urquhart and the Advanced Seeker Division of the Air Force Armament Laboratory, for their significant contributions of both computer resources and information about the image algebra. Finally, I wish to thank my wife, Conley, for her patience, caring, and understanding on those many days and nights when I could not escape my studies.

Christopher J. Titus

Table of Contents

List of Figures

List of Tables

Abstract

This study produced an image processing algorithm development tool on a VAX computer system using the recent advances in an Image Algebra developed by G. Ritter et al at the University of Florida. The image algebra provides the basis for a hardware and software independent environment for the expression of practically all image processing algorithms.

The goals of this project were twofold. The first goal was the implementation of the image algebra operators in a high level language to achieve hardware independence. The second goal was the design and implementation of a flexible preprocessor that could translate image processing algorithms, written in the image algebra language, into a high level computer language which could be compiled and executed on the VAX computer.

The implementation was achieved in the PASCAL computer language. All of the basic image algebra operators and the preprocessor were successfully programmed on the VAX computer, but a complete software independence of the image algebra was not achieved. This version also produces very large blocks of executable code for relatively simple algorithms.

Examples of the power and simplicity of the image algebra langauge and preprocessor environment are included.

A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA

I. Introduction

Background

Developers of image processing algorithms in military, industrial, and academic organizations have built large computer programs to facilitate their development work. Each program is the result of a constant evolution of image processing operations within each organization. Unfortunately, in the process of building excellent "in-house" development tools, each organization has created an image processing environment different from every other. Some image processing operations are unique to one environment while others may be common to a number of environments. But, since there is no common mathematical basis for all of these operations, it is often impossible to analyze and compare one environment with another.

This situation creates two major problems for the customers of image processing software, especially for the Department of Defense: (1) increased costs through funding of research and development in several organizations separately producing similar or identical processes, and (2) difficulty in analyzing and comparing the performance of competing algorithms (1:1-2).

Realizing this situation would create incompatibility and unnecessary confusion between competing automatic target recognition (ATR) algorithms, the Air Force Armament Laboratory (AFATL) at Eglin AFB, Florida contracted the University of Florida to develop an Image Algebra (IA). The goal the Image Algebra Project is

> ...the development of a complete, unified algebraic structure that provides a common mathematical environment for image algorithm development, optimization, comparison, coding, and performance evaluation. (1:2)

The primary goal of the AFATL was a concise mathematical structure for representing image transformations that was simple to learn yet powerful enough to perform any image processing operation. Toward this goal, the AFATL outlined a number of desirable properties that a useful image algebra should possess. These properties guided the development of the image algebra (1:3-4):

1. Basic operations that can model any image.
2. Elemental operations that can be combined to express any gray level image transformation.
3. Elemental operations that are few, simple, and easily learned by potential users.
4. Theorems that enable the simplification and optimization of algorithms through the use of identities involving the operators.
5. Notation that provides an understanding of image manipulations and is capable of suggesting new techniques.
6. Notation that allows programming languages to substitute concise image algebraic expressions for large blocks of code.
7. Notation that allows the use of libraries of image transformations.
8. Machine and language independence.
9. Compatibility with both sequential and massively parallel processors.

The result of this work by the University of Florida is an algebraic structure consisting of two operands, images and templates, and eight binary operators. Operations may occur between two images, an image and a template, or two templates. The result of these operations may be an image, a template, or a scalar.

Problem

The task of this thesis project is to implement the image algebra, as conceived by the University of Florida, on the VAX 11/780 located at the Air Force Institute of Technology. The implementation is to preserve the desirable properties of the image algebra outlined by AFATL and listed above. Consistency between the Air Force Institute of Technology image algebra (AFITIA) and the University of Florida image algebra (UFIA) is to be maintained.

Scope

The description of the image algebra continues to evolve at the University of Florida. Consequently, this project will not reflect the more recent changes to the structure. This implementation is based upon the description of the image algebra in the Image Algebra Tutorial and in the Final Report from Phase I of the Image Algebra Project (1,2).

This implementation is accomplished in the PASCAL computer language due to its availability and the experience

of the author. All of the PASCAL source code listed in the appendices was originally developed in TurboPascal (3), translated to VAX PASCAL (4), and ported to the VAX 11/780 at the Air Force Institute of Technology (AFIT).

General Approach

The general approach taken in this thesis project is the development of a computer program from the bottom up. This is contrary to accepted program development methods but it is desirable in this case. Consistency between the operands and operators of the UFIA description and AFITIA implementation is a primary concern in this project. Consequently, an implementation driven by the structure of the low level operands and operators is preferable.

The first task involves implementing the image algebra (IA) operands in PASCAL data structures and translating the IA operators to PASCAL procedures and functions. Based upon this design, the next task is the programming of an IA preprocessor that translates an IA description of any image processing operation into PASCAL source code for subsequent compilation. The final task is to install the image algebra routines and preprocessor into an environment on the VAX that allows a user to quickly implement any image processing operation.

Sequence of Presentation

The following report is divided into four major sections. Section II defines the basic operands and operators of the image algebra. Section III discusses the design of a general implementation of the image algebra to achieve a useful image processing development tool. Section IV covers the major part of this project: the design and implementation of an image algebra at AFIT. Section V compares three different image processing algorithms implemented in both PASCAL source code and the image algebra language. Finally, Section VI contains some general remarks on the accomplishments of this project and suggests some follow up projects. The appendices contain the source code to all of the AFIT image algebra programs.

II. Image Algebra

After studying the underlying operations of hundreds of existing image processing algorithms, the University of Florida investigators determined all of these routines could be performed with a small set of operands and operators. In fact, they proved this set of operands and operators, in conjunction with a programming language such as FORTRAN, is sufficient to perform any image-to-image transformation (1:48-62). This implies the image algebra operands and operators can be used to program all image transformations.

The next sections describe the image algebra operands, operators, and notation as developed by the University of Florida. These sections provide the reader with enough background to understand the subsequent sections of this report. The reader is directed to the Image Algebra Tutorial for a more complete mathematical description of the operands and operators (1:9-39).

Image Algebra Operands

The UFIA description contains six explicit operands: the set of real numbers, the set of complex numbers, any finite subset of k-dimensional space called $\underline{X}$, the power set on $\underline{X}$, the set of all images on $\underline{X}$, and the set of templates on $\underline{X}$. (1:6) Although these six operands completely define a

heterogeneous algebra, there are only two operands of primary interest in this project: images and templates.

Images are multiple dimension arrays of arbitrary integral size and may be integer, real, or complex valued. The points in images are referred to as pixels and each pixel in the array has a unique spatial identifier. In common image processing tasks, images represent two dimensional arrays of gray values describing scenes collected from vidicons, laser rangefinders, synthetic aperture radar, infrared imagers, or other sensing devices.

Images in the UFIA are denoted with capital letters from the beginning of the alphabet such as A, B, or C. Arbitrary pixels in n-dimensional images are depicted with n-dimensional vectors such as $\underline{k}$. That is, an arbitrary pixel in image A is denoted $A(\underline{k})$. The gray level at a particular image pixel is depicted with a lower case letter of the image designator: $a(\underline{k})$ represents the gray level at pixel $\underline{k}$ in image A (1:9).

To perform many image processing tasks, it must be possible to selectively choose and weight image pixels within small neighborhoods. The image algebra provides templates for this job. Templates, sometimes called windows or masks, are multiple dimension arrays of integral size and may be integer, real, or complex valued. Templates may be of arbitrary size and configuration but they are usually much smaller than images. Image algebra templates consist of two elements:

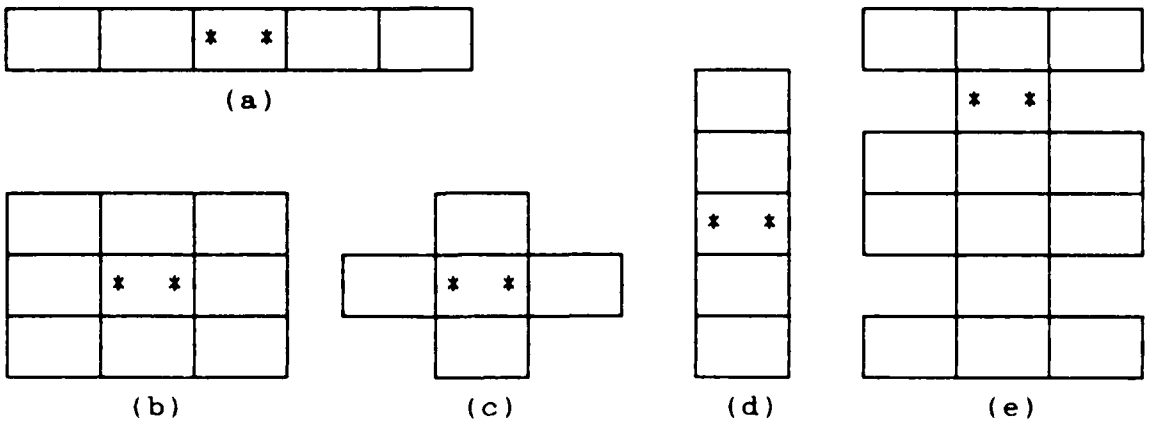


Figure 2.1 : Examples of Image Algebra Templates

(1) a configuration of pixels, and (2) a weight at each pixel in the configuration. The template configuration describes the relative orientation of all template pixels where the center pixel is commonly defined as the origin. The center pixel can be defined as any pixel in the configuration: it is not necessarily the physical center of the configuration. Only pixels in the template configuration may have non-zero values; any pixel not in the configuration has zero weight (1:23). Some examples of two dimensional templates are shown in Figure 2.1. The center of each template is marked with asterisks. For the purposes of the UFIA and this report, some of the template configurations have special names: Figure 2.1(a) is a horizontal template of size five, 2.1(d) is a vertical template of size five, 2.1(b) is a three-by-three Moore template (2:152), and 2.1(c) is a Von Neumann template of radius one (2:152).

The UFIA template notation is similar to the UFIA image notation. Templates are denoted with capital letters from the end of the alphabet such as R, S, or T. Arbitrary pixels in m-dimensional templates are depicted with m-dimensional vectors such as $\underline{y}$. For example, arbitrary pixel locations in template S are denoted by $S(\underline{y})$. The weight of a template pixel is represented by a lower case letter of the template designator: $s(\underline{y})$ denotes the weight at pixel $\underline{y}$ in template S (1:23).

There are two types of templates defined by the UFIA: variant and invariant. Variant templates may alter the weights of their pixels depending upon the location of the template in an image; that is, the pixel weights in variant templates are functions of the template position in an image. As the name implies, invariant templates are not allowed to alter their pixel weights during any operation with an image.

Image Algebra Operations

Due to various combinations of the two operands and eight operators of the image algebra, there are fourteen elemental operations as described by the University of Florida. The result of these operations can be an image, a template, or a scalar. There are unary and binary operations on both images and templates and binary operations between images and templates, but these fourteen operations are not necessarily the only ones supported by the image algebra. According to

the University of Florida,

> In general, all elementary functions supported by all the common higher level languages such as FORTRAN, PASCAL, etc. (i.e. sine, cosine, logs, exponential, etc.), are accepted operations of the image algebra. (5:15)

Figure 2.2 shows the names, notation, and mathematical description of the fourteen elemental operations in the image algebra. The vector $\underline{k}$ represents arbitrary pixels in an n-dimensional image, and the vectors $\underline{x}$, $\underline{y}$, and $\underline{z}$ represent arbitrary pixels in their associated m-dimensional templates. The ordered pair $(\underline{k},c(\underline{k}))$ represents an arbitrary image pixel and its associated gray value in the image C. Similarly, the ordered pair $(\underline{z},t(\underline{z}))$ represents an arbitrary template pixel and its associated weight in template T. The zero vector, $\underline{0}$, denotes the center pixel of a template. In the dot product operation, f is a scalar. The union of two images (A U B) or two templates (R U S) is defined as the image or template resulting from the union of all of the image or template pixels in the two images or templates. The intersection of two images ($A \cap B$) or templates ($R \cap S$) is defined as the image or template resulting from the intersection of all of the image or template pixels in the two images or templates.

Image-Image Operations. There are five elemental binary operations between images called image-image operations. Four of these operations (addition, multiplication, maximum, and exponentiation) result in another image. These operations are

Image-Image Operations

operation	notation	mathematical description
addition	$C = A + B$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = a(\underline{k}) + b(\underline{k})\}$ where C is defined for A U B
multiplication	$C = A * B$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = a(\underline{k}) * b(\underline{k})\}$ where C is defined for A ∩ B
maximum	$C = A \vee B$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = \max[a(\underline{k}), b(\underline{k})]\}$ where C is defined for A U B
exponentiation	$C = A **B$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = a(\underline{k}) ** b(\underline{k})\}$ where C is defined for A ∩ B
dot product	$f = A \cdot B$	$\equiv \{ f : f = \Sigma\, a(\underline{k})b(\underline{k})\}$ where C is defined for A ∩ B

Image-Template Operations

operation	notation	mathematical description
circle-plus	$C = A \oplus R$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = \Sigma\, a(\underline{k}+\underline{x}) * r(\underline{x})\}$ where C is defined for $A(\underline{k}) \cap R(\underline{0})$
circle-maximum	$C = A \circledvee R$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = \max[a(\underline{k}+\underline{x}) * r(\underline{x})]\}$ where C is defined for $A(\underline{k}) \cap R(\underline{0})$
square-maximum	$C = A \boxvee R$	$\equiv \{(\underline{k}, c(\underline{k})) : c(\underline{k}) = \max[a(\underline{k}+\underline{x}) + r(\underline{x})]\}$ where C is defined for $A(\underline{k}) \cap R(\underline{0})$

Template-Template Operations

operation	notation	mathematical description
addition	$T = R + S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = r(\underline{z}) + s(\underline{z});$ where T is defined for R U S
multiplication	$T = R * S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = r(\underline{z}) * s(\underline{z})\}$ where T is defined for R ∩ S
maximum	$T = R \vee S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = \max[r(\underline{z}), s(\underline{z})]\}$ where T is defined for R U S
circle-plus	$T = R \oplus S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = \Sigma\, r(\underline{x}) * s(\underline{y})\}$ where $\underline{z} = \underline{x} + \underline{y}$ and T is defined for $R(\underline{0})$ U $S(\underline{y})$
circle-maximum	$T = R \circledvee S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = \max[r(\underline{x}) * s(\underline{y})]\}$ where $\underline{z} = \underline{x} + \underline{y}$ and T is defined for $R(\underline{0})$ U $S(\underline{y})$
square-maximum	$T = R \boxvee S$	$\equiv \{(\underline{z}, t(\underline{z})) : t(\underline{z}) = \max[r(\underline{x}) + s(\underline{y})]\}$ where $\underline{z} = \underline{x} + \underline{y}$ and T is defined for $R(\underline{0})$ U $S(\underline{y})$

Figure 2.2: Image Algebra Elemental Binary Operations

Image A

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

Image B

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

Figure 2.3: Example of Image-Image Operations

performed on a pixel-to-pixel basis: a pixel of one image is added, multiplied, maximized, or exponentially multiplied with its spatially corresponding pixel in another image(1:12). For example, Figure 2.3 shows two images, A and B, of dimension 5x5. If addition (C = A + B) is performed on the two images, the resulting image is the sum of the two images at each pixel location: C(11) = A(11) + B(11), C(12) = A(12) + B(12), ..., C(55) = A(55) + B(55). Similarly, the image resulting from the maximum operation (C = A v B) on these two images would be: C(11) = max{A(11),B(11)}, C(12) = max{A(12),B(12)},..., C(55) = max{A(55),B(55)}. Multiplication and exponentiation are analogous operations.

The fifth elemental binary operation is the dot (inner) product. The result of this operation is a scalar, and it provides the image algebra with a method of mapping images to the real numbers. This operation is a sum of the product of

spatially corresponding pixels in two images. Using the two images from Figure 2.3, the result = A(11)*B(11) + A(12)*B(12) + ... + A(55)*B(55).

These five elemental image operations can be used to define other operations between images such as subtraction, division, and minimum (1:13).

Image-Template Operations. There are three elemental binary operations between an image and a template called image-template operations: circle-plus, circle-maximum, and square-maximum. The result of these operations is an image. Each pixel of the resultant image is a weighted function of the original image pixel and its neighbors delimited by the template configuration centered on this pixel (1:28). All template pixels that lay outside of the image boundaries at any image pixel are ignored.

The circle-plus operation performs a convolution: each pixel is the sum of the products of template pixels and image pixels delimited by the template configuration centered over the image pixel (1:28). For example, Figure 2.4 shows an image A of dimension 5x5 and template Y of dimension 3x3 with its center defined as y(5). If the circle-plus operation is performed between this image and template, one calculation of the operation is performed on the image pixel at row 3, column 4, shown as pixel A(34) in the figure. For the calculation at this pixel, template Y is overlaid on image A with the center

Image A

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

Template Y

1	2	3
4	* 5*	6
7	8	9

Figure 2.4: Example of Image-Template Operations

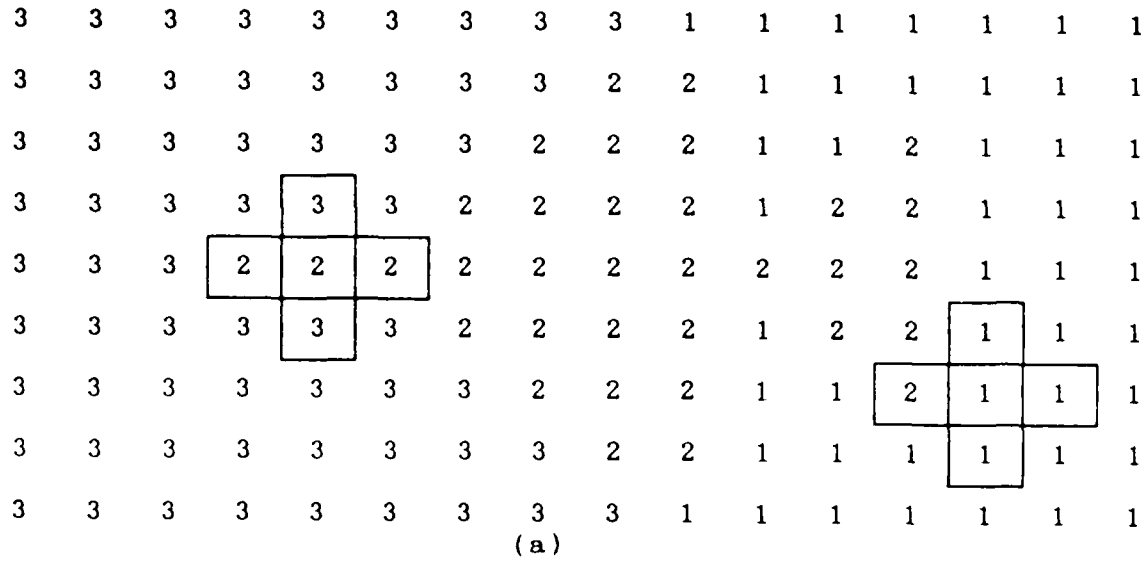
of the template, y(5), centered over pixel A(34). The operation computes the sum of the products of each template pixel and the image pixel it overlays. In this case, A(34) = y(1)*A(23) + y(2)*A(24) + y(3)*A(25) + y(4)*A(33)+ y(5)*A(34) + y(6)*A(35) + y(7)*A(43) + y(8)*A(44) + y(9)*A(45). This calculation is repeated for each image pixel to complete the entire circle-plus operation.

With the circle-maximum operator, each image pixel $\underline{x}$ is replaced with the maximum product of image pixel gray values within the neighborhood of $\underline{x}$ (defined by the template centered over $\underline{x}$) and the template pixel weights that overlay each image pixel. With the square-maximum operator, each image pixel $\underline{x}$ is replaced with the maximum sum of image pixel gray values within the neighborhood of $\underline{x}$ (defined by the template centered over $\underline{x}$) and the template pixel weights that overlay each image pixel. (1:28)

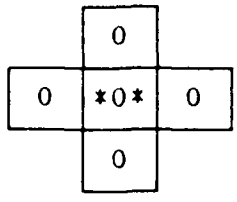
The circle-maximum and square-maximum operators can both perform gray level dilation depending upon the weights of the template. By choosing a template of unity weight for the circle-maximum operation and a template of zero weight for the square-maximum operation, a gray level dilation is performed. Figure 2.5(c) shows the result of dilating the image in 2.5(a) with the square-maximum operator and template shown in 2.5(b). As an example of the intermediate calculations in the square maximum operation, the template is shown centered over two pixels in 2.5(a). The result of the square-maximum operation for these two pixels, the maximum value after summing each template weight with the image pixel it overlays, is shown in 2.5(c).

These three elemental image-template operations can be used to define other operations between images and templates such as circle-minimum and square-minimum.

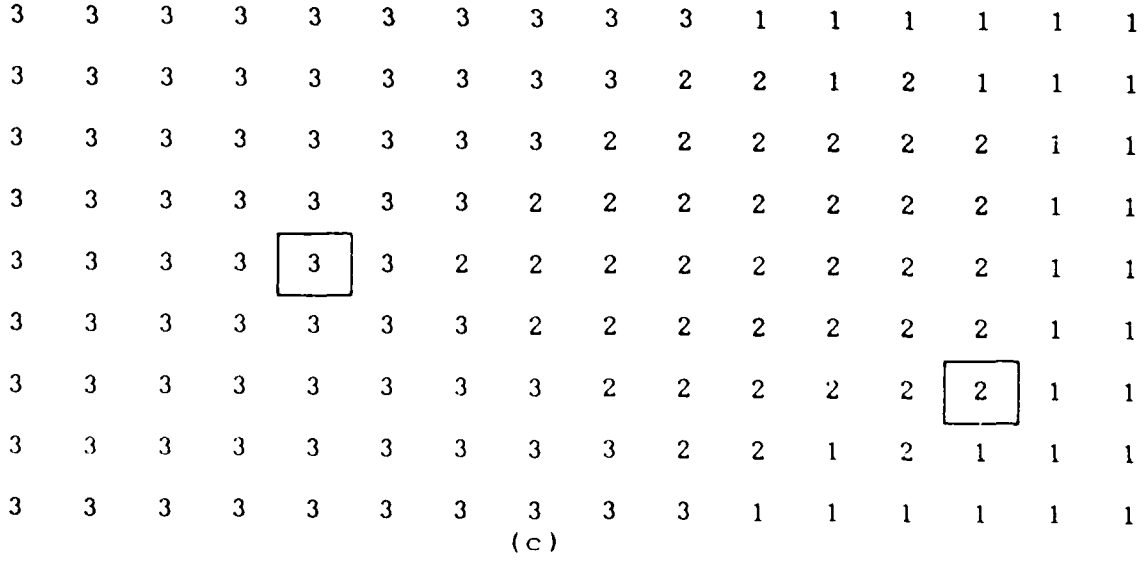
Template-Template Operations. There are six elemental binary operations between two templates called template-template operations: addition, multiplication, maximum, circle-plus, circle-maximum, and square-maximum. These operations are similar to their image-image and image-template counterparts; the major difference is the region over which the operations are defined. Addition and maximum operations are defined over the union of the two template configurations, but multiplication is defined only over the intersection of



(a)



(b)



(c)

Figure 2.5: Example of Gray Level Dilation

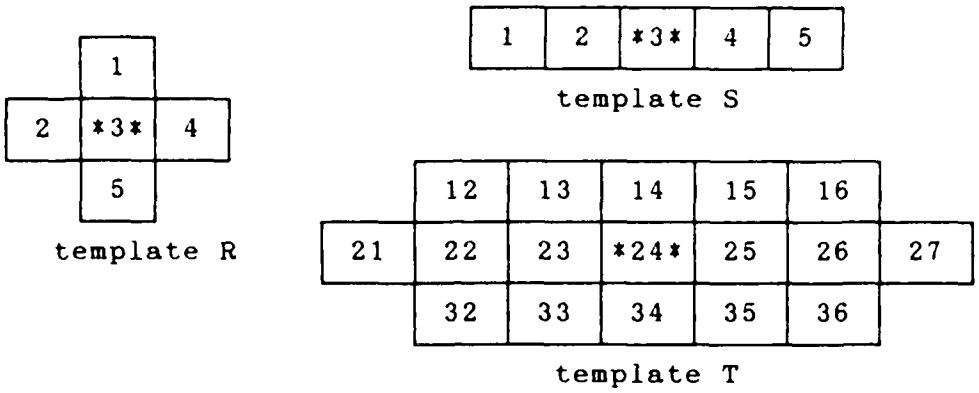


Figure 2.6: Template Configuration Resulting from T = R(+)S, R(v)S, or R[v]S

the two templates. Circle-plus, circle-maximum, and square-maximum operations are defined over the configuration formed by the union of the pixels of one template centered over all of the pixels of the other template (1:30). For example, Figure 2.6 shows a Von Neumann template, R, with its center defined as r(3), and a horizontal template, S, with its center defined as s(3). The template resulting from the circle-plus, circle-maximum, and square-maximum operations is shown as template T with center t(24). Template T is formed by the union of the configurations constructed by overlaying the center pixel of template R on each pixel of template S: $T(\underline{x}) = R(\underline{0}) \cup S(\underline{y})$. In this example, the pixels at t(21), t(12), t(22), t(32), and t(23) are the result of pixels r(1) through r(5) when template R is centered at pixel s(1).

The computation of the resultant weights in template T under the operations of addition, multiplication, and maximum

is simple. Since template addition and maximization are performed over the union of the two templates, the resulting template is the addition or maximum of pixels from each template with the same configuration coordinates. Template multiplication is performed over the intersection of the two templates: each resulting pixel is the product of a pixel from each template with the same configuration coordinates (1:25). Recall that template pixels outside the configuration have weights of zero.

The computation of template T resulting from the circle-plus, circle-maximum, and square-maximum operations between templates R and S is more difficult. Conceptually, template T is the result of two operations between the operand templates R and S. First, the template T configuration is computed from the union of template R centered over each cell of template S, as discussed above. Second, the weight of each cell in the configuration of template T is computed. For convenience, designate each cell of templates R, S, and T as R(ri,rj), S(si,sj), and T(ti,tj) respectively, where (ri,rj), (si,sj), and (ti,tj) denote the cell's offset from the center of the template. The center is designated R(0,0), S(0,0), and T(0,0). Thus, the weight of each cell in T can be computed as follows for each elemental operation:

(1) circle-plus: $T(t_i,t_j) = \Sigma [R(r_i,r_j)*S(s_i,s_j)]$ where $(t_i,t_j) = (r_i+s_i,r_j+s_j)$ and $R(0,0)$ is centered over $S(s_i,s_j)$

(2) circle-maximum: $T(t_i,t_j) = \max[R(r_i,r_j)*S(s_i,s_j)]$ where $(t_i,t_j) = (r_i+s_i,r_j+s_j)$ and $R(0,0)$ is centered over $S(s_i,s_j)$

(3) square-maximum: $T(t_i,t_j) = \max[R(r_i,r_j)+S(s_i,s_j)]$ where $(t_i,t_j) = (r_i+s_i,r_j+s_j)$ and $R(0,0)$ is centered over $S(s_i,s_j)$

A more rigorous mathematical description of these operations can be found in the Image Algebra Tutorial (1:30).

These six elemental template operations can be used to define other operations between templates such as subtraction, division, minimum, circle-minimum, and square-minimum (1:25).

Algorithm Optimization

Although the convolution operations between two templates are complicated and appear useless in real image processing algorithms, they form a very important part of the IA. These convolution operators are important for the composition and decomposition of templates which provide tools for program optimization. (1:39)

The decomposition of templates can be used to break a large template into an equivalent set of smaller templates which yield a decrease in the computation of an expression. For example, the circle-plus operation can be used to calculate

the local average gray level in an image (dimension 256x256) with a template (dimension 3x3) through $C = A \oplus R$. If template R is decomposed into two smaller templates, S (dimension 3x1) and T (dimension 1x3), then the average image can be calculated by $C = A \oplus R = A \oplus (S \oplus T) = (A \oplus S) \oplus T$. This equation shows a number of methods for calculating the average, each one with a very different computational load. Since each pixel in $C = A \oplus R$ requires the summation of nine products, the entire operation needs 589,824 multiplications and 524,288 additions. However, since each pixel in $C = (A \oplus S) \oplus T$ requires the summation of three products for each circle-plus operation, this entire operation requires 393,216 multiplications and 262,144 additions. Thus, the second formulation needs 30 percent fewer multiplications and 50 percent fewer additions: template decomposition can provide algorithm optimization.

III. General Implementation of the Image Algebra

The ultimate goal of the image algebra project is a standardization of the image processing development tools used by many different companies and agencies around the country. Because each organization has a lot of money invested in its own image processing tools, the only viable way of obtaining standardization is by convincing each group that the image algebra is an efficient and powerful development tool.

The image algebra developed by the University of Florida, and briefly presented in Section II of this report, provides a strong foundation for an image processing environment that could become a standard. The mathematical structure and small number of elemental operators allow a user to model any image, perform any gray level transformation, and optimize algorithms through identities without forcing the user through a long learning process. Additionally, the notation provides an understanding of the basic operations involved in any image processing algorithm described with the image algebra.

In essence, the University of Florida has achieved the first six goals of a desirable image algebra outlined by the Air Force and discussed in Section I. They have succeeded in developing an image algebra that is both powerful and concise. But, if the image algebra is to become a desirable development tool, its implementation must retain this power and brevity as

well as promote a complete image processing environment. The implementation of the image algebra is as important to its acceptability as its mathematical structure and power. Thus, what attributes must the image algebra implementation possess in order to become a useful and desirable image processing development tool?

Image Algebra Notation

The implementation of the image algebra notation should be concise and flexible. Increases in the production of image processing algorithms can be realized through a notation that allows programmers to represent complicated operations with single character operators. In addition, the notation must allow a programmer the flexibility to manipulate the image algebra operators as if they were ordinary algebraic operators in a computer program.

The image algebra notation must provide access to the program statements supported by many high level languages. Program statements are those structures that provide program control during execution of the routine. They include assignment (A=B), conditional branching (if-then-else), and repetitive looping (for-do, while-do) statements. Because each computer language supports and implements these program structures differently, the image algebra notation must support program statements suitable for translation to any computer language.

The implementation should support arbitrary names for image, template, and scalar operands, and program variables. Operand and variable names should not be limited to certain lengths or certain letters of the alphabet.

The notation must support all of the elemental operations of the image algebra, and the computer representation of each operator should closely resemble its written representation. A user should be able to enter an image processing algorithm into a computer exactly as it is written on paper.

Computer and Language Independence

One of the objectives of a standardized image algebra processing environment is the ability to transfer image processing algorithms from one computer system to another without (ideally) modification. The purpose of this goal is to provide users of the image algebra with the ability to share image processing routines without forcing everyone to enter each algorithm by hand. The preferred method would be tape transfers between computer sites. This goal requires that the image algebra environment be machine and language independent.

There are three basic approaches for making the image algebra both machine and language independent. First, require every user of the image algebra to employ identical computer hardware and software. Second, implement the image algebra in a high level language. Third, design a high level language,

based upon the image algebra notation, that can be translated to any computer system. Because the ultimate goal of the image algebra project is standardization of the entire image processing community, the third method is the most logical approach.

The first two approaches present problems to wholesale acceptance of the image algebra. Foremost, it is unlikely that any organization would significantly modify its large investment in computer resources to accommodate a new program. High level languages approach independence, but no high level language is completely machine independent. Further, a choice of one language or another will present compatibility problems to one group or another because they do not use that specific language.

The key to general acceptance of the image algebra is an environment that can interface the image algebra description of an algorithm to the present computer resources of each organization. This interface would allow the sharing of image processing routines by translating image algebra descriptions into a language supported by that organization's computer system. Subsequent compilation of the translated routine would allow its execution on the host computer without modification.

The real benefit of this design is derived through the flexibility of tailoring the image algebra operations to each

organization's computer system while retaining machine and language independence through the interface. This allows each organization to optimize the IA operations while maintaining complete IA language compatibility.

Image Algebra Preprocessor

The interface is provided by a preprocessor that translates image algebra expressions into equivalent blocks of code in FORTRAN, PASCAL, ADA, or some other language. The translated code is subsequently compiled and linked with libraries of image algebra operations into executable modules.

The requirements for implementing the image algebra with this type of interface are minimal. Each organization needs both a high level language description of the image algebra operations and a preprocessor. Each organization tailors the description of the image algebra operations to their specific computer system and then places them into the computer's libraries. Once the preprocessor and operators are installed, the computer can be programmed to automatically translate, compile, and link each routine into an executable module.

This design requires the image algebra preprocessor to be flexible. The preprocessor must be capable of translating image algebra expressions and other program statements into equivalent code in several different languages. This does not imply that every preprocessor must be capable of translating image algebra notation into more than one language. Each

computer system will employ a preprocessor that translates the image algebra expressions and program statements into the high level language(s) supported by that computer.

It is anticipated that image processing algorithms will possess statements containing multiple operations per line. A scalar algebraic equation like c = a * b - (c + d)/e has an equivalent representation in the image algebra as C = A * B - (C + D)/E. Rather than forcing the image processing engineer to write this equation as a series of statements with one operation per line, it would be desirable to enter the entire equation on one line. Consequently, the preprocessor should be able to translate multiple image algebra expressions per line.

There are a number of ways to implement the image algebra preprocessor. One method is a preprocessor that translates each IA expression into the equivalent source code of a high level language by directly substituting blocks of code for each IA expression. This method allows each program module to be self-contained (free from external procedures) but each program module may contain redundant blocks of code. This method creates modules that are likely to be larger than necessary, but they may execute more rapidly because they avoid the inherent overhead of procedure calls.

Another method, the one implemented in this project, is a preprocessor that translates each IA expression into a series

of equivalent high level language procedure calls. Although each program module must be linked with the library of image algebra operations, it is a self-contained executable module that prevents redundant blocks of code. This method creates modules that are relatively compact but they execute more slowly because they rely almost entirely upon procedure calls. The major advantages of this technique are the simple design of the preprocessor and the standard interface between the

The operation of the preprocessor implemented at AFIT in this project is demonstrated by Figures 3.1 and 3.2. Figure 3.1 shows an IA source file containing a number of operations. The preprocessor translates the IA expressions into a series of PASCAL procedure calls as shown in Figure 3.2. At this point, it is not necessary for the reader to understand all of the notation contained in these two figures. That will be explained in the next section.

```
Image_and_Template_Operations

 (*   This is the program comment section.  Each line is     *)
 (* marked with the PASCAL comment delimiters at the ends.   *)
 (* The program comment section in the IA file is not        *)
 (* passed to the PASCAL source file.                        *)
 (*   The following lines contain the other IA program       *)
 (* sections.  The program code section includes two         *)
 (* template definitions.                                    *)
 (* constant declaration section *)
 const MaxLoop = 3
 (* type declaration section *)
 type itype = integer
      ttype = integer
 (* variable declaration section *)
 var R,S,T : template
     A,B,C : image
     j : integer
 (* program code section *)
 begin
  (* define templates S and T *)
  invariant template S
   begin
   S(x,y)   = 4
   S(x,y+1) = 5
   end
  invariant template T
   begin
   T(x,y-1) = 1
   T(x,y)   = 2
   T(x+1,y) = 3
   end
  R := S (+) T
  GetImage (A,'')
  for j := 1 to MaxLoop do
   (* loop the statements delimited by "begin" and "end" *)
   begin
    B := A>=j
    C := (C + B) [v] R
    end
  PutImage (C,'')
  end.
```

Figure 3.1: Sample Image Algebra File

```
[inherit('iiaoper.env','iio.env')]
program image_and_template_operations (input,output);
 const maxloop = 3;
 var r,s,t : template;
     a,b,c : image;
     j : integer;
 begin
  reset (input);
  rewrite (output);
  s.fig(.1.).r:= 0;  s.fig(.1.).c:= 0;  s.fig(.1.).w:=4;
  s.fig(.2.).r:= 0;  s.fig(.2.).c:=+1;  s.fig(.2.).w:=5;
  s.num := 2;
  t.fig(.1.).r:= 0;  t.fig(.1.).c:=-1;  t.fig(.1.).w:=1;
  t.fig(.2.).r:= 0;  t.fig(.2.).c:= 0;  t.fig(.2.).w:=2;
  t.fig(.3.).r:=+1;  t.fig(.3.).c:= 0;  t.fig(.3.).w:=3;
  t.num := 3;
  (***** r:=s(+)t *****)
  TempCirclePlus (s,t,tR1);
  r := tR1;
  getimage (a,'');
  for j := 1 to maxloop do
   begin
    (***** b:=a>=j *****)
    maxval := MaxValImage (a);
    CharImage (a,j,maxval,iR1);
    b := iR1;
    (***** c:=(c+b)[v]r *****)
    ImageAdd (c,b,iR1);
    ImageTempSquareMax (iR1,r,iR2);
    c := iR2;
    end;
  putimage (c,'');
  end.
```

Figure 3.2: PASCAL Source Code After Preprocessing the Sample Program of Figure 3.1

IV. AFIT Implementation of the Image Algebra

The image algebra, as conceived by the University of Florida, can be the basis for a powerful image processing development environment. It has the capabilities to perform any gray level image transformation, and its notational and operational simplicity are surprising. This image algebra has the potential to become a widely accepted image processing tool.

As discussed in Section III, the most critical component of the image algebra will be its computer implementation, especially the user interface. If the image algebra is to gain wide spread acceptance, it must have a user interface that is both simple and capable of utilizing the full power of the image algebra.

This section discusses the design and implementation of the image algebra at AFIT. The goal is an image processing development tool with a simple user interface that allows full access to the power of the image algebra. This implementation is intended to achieve three of the desirable properties of a useful image algebra outlined in Section I: (1) substitution of concise image algebra expressions for large blocks of program code, (2) notation that allows the use of libraries of image transformations, and (3) machine and computer language independence.

Development Environment and File Structure

Before discussing the explicit design of the AFIT image algebra, it is helpful if the reader understands the overall AFITIA structure and how it is incorporated into the VAX 11/780 processing environment using VAX PASCAL.

The major components of the AFITIA development tool are derived from five PASCAL source code files: (1) two image algebra operations files named iIAOPER.PAS and rIAOPER.PAS, (2) two input/output operations files named iIO.PAS and rIO.PAS, and (3) the preprocessor file named PREPROC.PAS. The first two files contain the source code for all of the operand type declarations and operations implemented in the AFIT image algebra with integer and real operands. The second two files contain the source code for all of the input/output operations with integer and real operands. Two files each for the IA operations and input/output routines are required because of the strong data typing used in PASCAL: some IA operations can not be programmed in PASCAL to manipulate both real and integer operands. Therefore, a PASCAL implementation of the IA requires a set of IA operations each for real and integer operands. The "i" and "r" preceding the IA operation and input/output file names above denote the type of operands, integer or real, used in that file's procedures. A copy of the source code for the AFITIA operations with real operands is in Appendix A, and a copy of the source code for the AFIT

input/output operations with real operands is in Appendix C. The fifth PASCAL file contains the AFITIA preprocessor used for translating image algebra expressions into a series of PASCAL procedure and function calls. A copy of the source code for the AFITIA preprocessor is contained in Appendix B.

Before any translation of image algebra expressions can be accomplished on the VAX 11/780, the development environment must possess eight files derived from the above source code files. The first two are the object code descriptions of the image algebra operations, iIAOPER.OBJ and rIAOPER.OBJ, created by compiling iIAOPER.PAS and rIAOPER.PAS, respectively. The next two are the object code descriptions of the input/output operations, iIO.OBJ and rIO.OBJ, created by compiling iIO.PAS and rIO.PAS, respectively.

The other four files, iIAOPER.ENV, rIAOPER.ENV, iIO.ENV, and rIO.ENV, contain the procedure declarations from the image algebra operation and input/output files. These files are required to ensure proper compilation of a translated IA file because PASCAL requires that all procedures and functions be declared to the compiler before they are used. This method permits VAX PASCAL to separately compile and link IA source files, IA operation files, and input/output files. Like the object code files, these files are created during compilation of the PASCAL source code files by using the "environment" compiler attribute in VAX PASCAL.

These eight files plus an executable version of the preprocessor are all that is necessary to translate an image algebra description of any image processing algorithm into an executable module. The AFITIA development tool uses a command file called TIA.COM to automate this translation process. A copy of this command file is in Appendix D. Figure 4.1 shows a block diagram of the entire AFITIA development tool, the interrelationship between all of the files used to build an executable image processing module, and the flow of execution in the automatic translator command file. The execution of this command file is outlined below.

Starting at the top, the file containing the image algebra language description, the IA file, is copied to an intermediate working file named TRANSLAT.IA. The name of the file may be any valid filename, and it is assumed to have a ".IA" extension.

Next, the preprocessor is invoked to translate the IA expressions in this working file into PASCAL source code. First, the entire TRANSLAT.IA file is converted to lower case letters, all comments are removed, and the result is placed in TRANSLAT.TMP. Second, the preprocessor examines TRANSLAT.TMP and determines which type of the image and template operands, integer or real, are used in this routine. The default data type is real. Based upon this determination, the preprocessor writes a statement to another working file, TRANSLAT.PAS, that

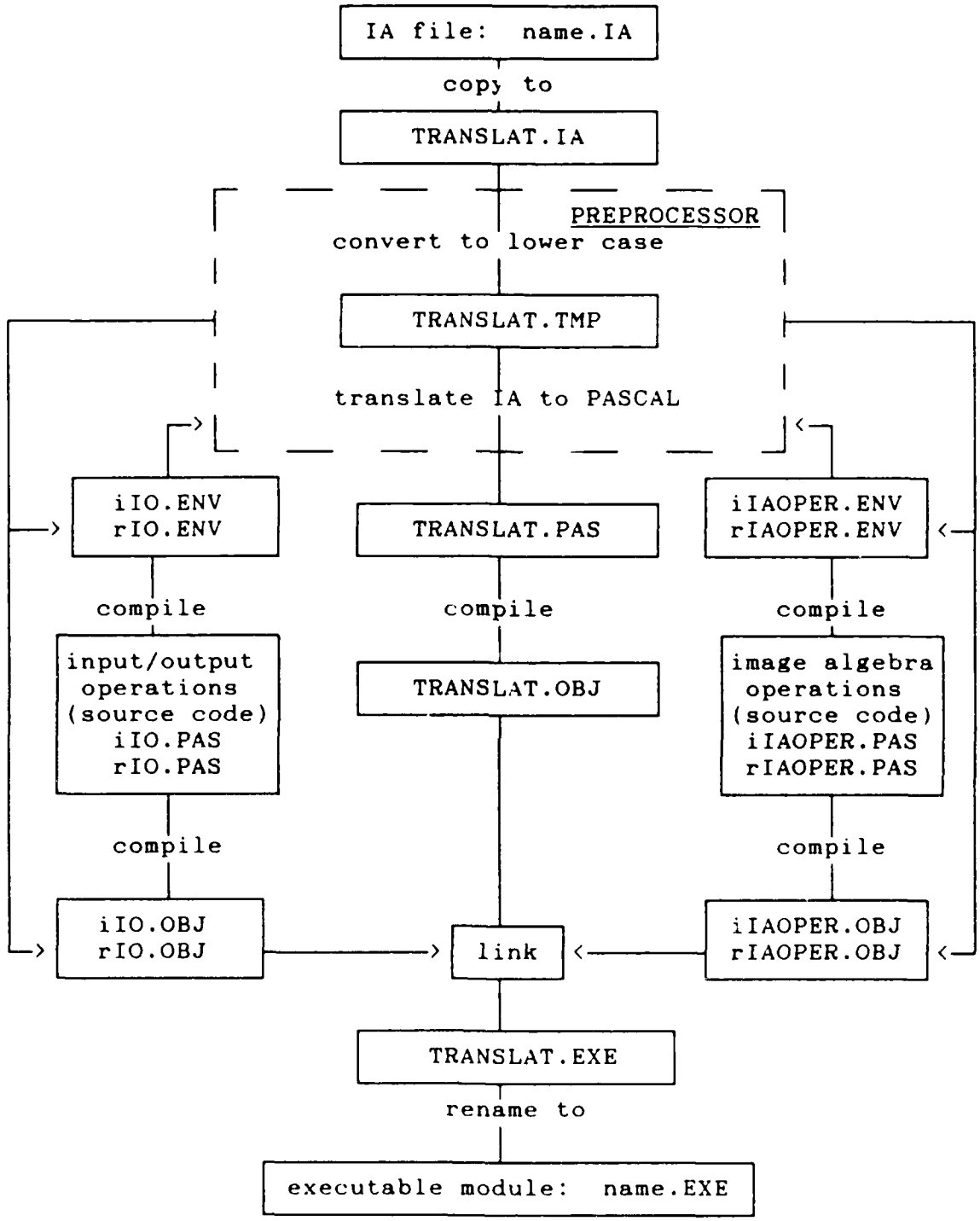


Figure 4.1: AFITIA Development Environment and File Structure

allows the inclusion of the proper environment files, iIO.ENV or rIO.ENV, and iIAOPER.ENV or rIAOPER.ENV, for compilation. Finally, the preprocessor continues examining TRANSLAT.TMP, translates each line of the file containing IA expressions, and outputs the translated code to TRANSLAT.PAS.

Once the entire algorithm is translated, the PASCAL compiler is invoked to operate on the TRANSLAT.PAS file, and the compiled (object) code is returned in TRANSLAT.OBJ. The object file is subsequently linked with the proper image algebra operation and input/output object files resulting in an executable file named TRANSLAT.EXE. This executable file is then renamed to the file name of the original IA file with a ".EXE" extension, and all of the TRANSLAT.*** files are deleted from the directory.

After the executable file is automatically built by the process described above, the algorithm may be run from the computer's command line.

AFIT Image Algebra Language

One of the goals of both the UFIA and this project was the concise expression of image processing algorithms. Given the profusion of image processing algorithms with controlled execution, differing data structures, and diverse operations, this is a formidable task. However, the AFITIA language is designed to overcome these obstacles. First, the AFITIA notation supports the controlled execution of IA expressions.

Second, the AFITIA operands are flexible enough to accommodate many different image and neighborhood configurations. Third, the AFITIA can perform any image transformation because its basic operations are identical to those of the UFIA. Recall that the University of Florida proved this set of operations was capable of performing any image-to-image transformation when used in conjunction with controlled program execution. The following sections discuss each of these aspects of the AFITIA language in more detail.

Program Control. The UFIA elemental operations described in Section II can serve as the basis for an IA language, but these operations alone cannot form a complete IA language capable of describing all transformations. Many image processing algorithms require additional program statements, such as conditional branching and repetitive looping, to control the execution of an algorithm. Consequently, a complete image algebra language must support both the operations defined in the UFIA and the programming statements found in the more popular high level computer languages.

Since the image algebra will be used with many different computer languages, the image algebra language must define and support program control statements that can be translated into any computer language. Designing a language to be independent of the target language of the preprocessor will be necessary because different computer languages support different control

statements. For example, PASCAL supports "if-then-else" and "case" conditional statements and "for", "while" and "repeat" repetitive statements; FORTRAN has "if-then-else", "do", and "do-while" statements; the C language supports "if-then-else", "while", "for", "do-while", and "switch" statements; and PL/I has "if-then-else", "select", "do", "while", and "until" control statements. (3:57-61; 7:9-3,9-9,9-12; 8:67-85,151-174; 9:88-101,103-127) Although there is a common set of control statements, there are differences that make it difficult to implement a universal notation.

Therefore, this project did not attempt to design generic control statements that could be translated to equivalent statements in any high level computer language. Instead, the AFITIA language retains the PASCAL constructs in order to form an image algebra language that supports repetitive looping and conditional branching. Consequently, this implementation of the image algebra language is not completely machine or language independent, but it does progress in that direction.

Operand Data Structures and Notation. The operands of the AFITIA must be capable of modeling and transforming any practical image. Consequently, the data structures of the operands must be flexible enough to handle many different image formats and neighborhood configurations.

In the UFIA, the number of images and templates, and their respective dimensions, are limitless. Unfortunately, an

implementation of the image algebra on a computer requires some compromises. Implementing the image algebra on the AFIT computer requires that the dimensions of the UFIA operands be restricted because of limited computer resources. The size of the operands supported by the AFITIA should be sufficient for most image processing tasks.

In this version of the AFITIA, images are limited to two-dimensional arrays, and the number of images available to the user is software limited to 100. Two-dimensional arrays are used to minimize the memory and program requirements. Images may be named with any combination of characters (except v), numerals, and underscores up to eighty characters long. A software switch, iType, included in the IA source file can be used to implement the images with real or integer gray values. The default switch setting is real. Complex valued images are not supported in this version of the AFITIA. Each dimension of the image may be any size up to a maximum value set by the user with two other software switches found in the IAOPER.PAS files, MaxImageRow and MaxImageColumn.

The data structure of the image operand is a record with three fields: row dimension, column dimension, and an array of integer gray levels indexed by row and column position. This data structure allows the AFITIA to manipulate multiple, arbitrarily sized images. Additionally, since the dimensions of the image can be initialized by a procedure that reads in

the input image, the user is freed from tracking all of the image dimensions during execution.

The image notation used in this project and by the University of Florida is similar. Images are denoted with capital letters from the beginning of the alphabet such as A, B, and C. Arbitrary pixels in an image are represented with the vector notation $\underline{k}$, and the gray value at an arbitrary pixel in image A is denoted by $A(\underline{k})$. Specific pixels in the two-dimensional AFITIA images are denoted with row and column qualifiers to the image letter designator such as A(2,3) or B(7,4).

In this version of the AFITIA, one-dimensional arrays are used for storing template configurations and weights. One-dimensional arrays were chosen for the template operands in order to simultaneously increase configuration flexibility and minimize computer storage requirements. Like image operands, templates may be named with any combination of letters (except v), numerals, and underscores up to eighty characters in length. A software switch, tType, included in the IA source file can be used to implement the templates with real or integer weights. The default switch setting is real. Complex valued templates are not supported in this version of the AFITIA. The size of the templates is presently limited by another software switch, MaxTempCell, in the IAOPER.PAS file

to 100 pixels in each configuration. All pixels not within the configuration are defined to have a value of zero.

The data structure of the template operand is a record with two fields. The first field contains the number of pixels in the configuration. The second field is an array with each array cell representing a pixel in the template configuration. Each array cell contains a record with three fields of information about that pixel: the (1) row and (2) column offset from that template's center pixel, and (3) the weight of that pixel. This data structure permits arbitrary row and column offsets for each pixel and, therefore, allows arbitrary template configurations to be implemented while minimizing the computer storage requirements.

The template notation used in this project and by the University of Florida is similar. Templates are denoted with capital letters from the end of the alphabet such as R, S, and T. Arbitrary pixels in a template are represented with the vector notation $\underline{z}$ where $\underline{z}$ stands for the m-dimensional offset from the template's center pixel. (Recall that the center pixel is not necessarily the physical center of the template's configuration.) The weight at an arbitrary pixel in template R is denoted by $R(\underline{z})$. Specific pixels in AFITIA template configurations are represented by row and column offsets from the template's center pixel. For example, R(-2,3) is a pixel two rows above and three columns to the right of the center

pixel, and the center pixel of template S may be represented by S(0,0) or S(0).

Operator Implementation and Notation. The five elemental image-image operations, three elemental image-template operations, and six elemental template-template operations of the UFIA are preserved in the AFITIA. Each operation is implemented as a PASCAL procedure or function. Each basic operation is performed by calling the appropriate subroutine, passing the necessary operands, and obtaining the result returned from the subroutine. The source code for all of the AFITIA operations is in Appendix A.

This implementation of the operators directly supports the concise notation feature described above. The large blocks of code needed to program each operation are replaced by a short subroutine call.

The interface between the program calling the procedure and the procedure itself is designed to provide maximum program flexibility while ensuring data and variable integrity. Using the parameter passing schemes provided by PASCAL, each procedure obtains its operands through value parameters and the results are returned to the calling program through variable parameters. Local variables are used in each procedure to prevent the procedures from modifying any variables not within their scope. This scheme allows each procedure to perform its operation without modifying the data

in either operand, and it allows the result to be returned to an arbitrary variable name in the calling program.

As noted above, the AFITIA is comprised of a collection of PASCAL procedures and functions that allow the substitution of short subroutine calls for the computer code of each elemental operation. This attribute partially supports the sixth image algebra guideline outlined in Section I. However, requiring a user to write image processing algorithms as a collection of procedure calls is not compatible with the goal of writing algorithms using the image algebra notation.

Writing computer programs in the image algebra notation requires a mapping of image algebra operator symbols to ASCII characters. Because many of the operator symbols in the image algebra cannot be represented by a single ASCII character, the AFITIA uses short strings of characters that closely resemble the image algebra notation. Figure 4.2 displays all of the operators, their UFIA notation, and their equivalent AFITIA notation. In the figure, A, B, and C are defined as images, R, S, and T are defined as templates, and f is defined as a scalar. Notice the desirable similarity between the operator symbols of the UFIA and AFITIA. This scheme avoids memorizing a mapping between image algebra operator symbols and ASCII characters. Additionally, this should ease the learning necessary to write computer programs using the image algebra.

Image-Image Operations

operation	UFIA notation	AFITIA notation
addition	C = A + B	C = A + B
subtraction	C = A - B	C = A - B
multiplication	C = A * B	C = A * B
division	C = A / B	C = A / B
maximum	C = A ∨ B	C = A v B
minimum	C = A ∧ B	C = A ^ B
exponential	C = A **B	C = A **B
dot product	f = A • B	f = A . B

Image-Template Operations

operation	UFIA notation	AFITIA notation
circle-plus	C = A ⊕ R	C = A (+) R
circle-maximum	C = A ⊗ R	C = A (v) R
circle-minimum	C = A ⊘ R	C = A (^) R
square-maximum	C = A ⊠ R	C = A [v] R
square-minimum	C = A ⊡ R	C = A [^] R

Template-Template Operations

operation	UFIA notation	AFITIA notation
addition	T = R + S	T = R + S
subtraction	T = R - S	T = R - S
multiplication	T = R * S	T = R * S
division	T = R / S	T = R / S
maximum	T = R ∨ S	T = R v S
minimum	T = R ∧ S	T = R ^ S
circle-plus	T = R ⊕ S	T = R (+) S
circle-maximum	T = R ⊗ S	T = R (v) S
circle-minimum	T = R ⊘ S	T = R (^) S
square-maximum	T = R ⊠ S	T = R [v] S
square-minimum	T = R ⊡ S	T = R [^] S

Other Operations

operation	UFIA notation	AFITIA notation
assignment	A = B	A := B
	R = S	R := S
characteristic images (image-image)	A = B	A = B
	A > B	A > B
	A ≥ B	A >= B
	A < B	A < B
	A ≤ B	A <= B
	A ≠ B	A <> B
characteristic images (image-scalar)	A = f	A = f
	A > f	A > f
	A ≥ f	A >= f
	A < f	A < f
	A ≤ f	A <= f
	A ≠ f	A <> f
absolute value	|A|	|A|
	|R|	|R|

Figure 4.2: AFIT Image Algebra Operator Syntax

AFIT Image Algebra Preprocessor

If the image algebra notation is to be used for writing computer programs, then an interface between the image algebra language and the host computer language must be provided. This interface must be capable of translating image algebra operands and operator symbols into the more popular high level computer languages.

Consequently, the final component of the AFIT image algebra environment is a preprocessor that translates image algebra expressions into to a sequence of PASCAL procedure calls required to execute the algorithm. This preprocessor allows a user to write an image processing algorithm entirely with image algebra operands, operators, and common programming constructs, and with little knowledge of computer programming syntax. The source code for the AFITIA preprocessor is in Appendix B.

The AFITIA preprocessor is designed to translate any unambiguous image algebra expression, using the notation shown in Figure 4.2, into an equivalent sequence of PASCAL procedure calls. This preprocessor also supports any of the PASCAL programming constructs such as assignment, branching, and repetitive looping. The development of an image processing algorithm using the AFIT image algebra notation with the AFIT preprocessor requires the user to follow a few guidelines when writing the program.

AFIT Image Algebra Syntax

The AFITIA preprocessor expects the image algebra file to be comprised of up to six basic sections: (1) program name, (2) program comment, (3) constant declaration, (4) type declaration, (5) variable declaration, and (6) program code. Each section is delimited by a keyword or symbol, but all sections need not be present to form a working algorithm. Only the sections necessary for proper compilation are required. Additionally, the order of the sections in the image algebra file is very flexible. The only requirements are that the program name, if it is included, must be first section and the program code must be last section. Any comments embedded in the IA file are ignored. Figure 4.3 shows an IA program with all six sections included.

Program Name. An image algebra program may be named in the IA source file, but it is not required. The preprocessor interprets the first non-blank line of the file as the program name unless the first line contains a comment delimiter or a word reserved to head any of the major file sections. In the latter case, the program is assumed to be unnamed.

Program Comment. The comment section is an optional section for all of the comments the user desires to imbed in the IA file about the program's operands, execution, input, and output. Each comment line is delimited by "(*" on the left end and "*)" on the right end. The length and content of

```
Image_and_Template_Operations

 (*   This is the program comment section.  Each line is     *)
 (* marked with the PASCAL comment delimiters at the ends.   *)
 (* The program comment section in the IA file is not        *)
 (* passed to the PASCAL source file.                        *)
 (*   The following lines contain the other IA program       *)
 (* sections.  The program code section includes two         *)
 (* template definitions.                                    *)

 (* constant definition section *)
 const MaxLoop = 3

 (* type definition section *)
 type itype = integer
      ttype = integer

 (* variable declaration section *)
 var R,S,T : template
     A,B,C : image
     j : integer

 (* program code section *)
 begin
  (* define templates S and T *)
  invariant template S
   begin
   S(x,y)   = 4
   S(x,y+1) = 5
   end
  invariant template T
   begin
   T(x,y-1) = 1
   T(x,y)   = 2
   T(x+1,y) = 3
   end
  R := S (+) T
  GetImage (A,'')
  for j := 1 to MaxLoop do
   (* loop the statements delimited by "begin" and "end" *)
   begin
    B := A>=j
    C := (C + B) [v] R
    end
  PutImage (C,'')
  end.
```

Figure 4.3: Sample Image Algebra File

the program comment section is arbitrary because the AFIT preprocessor ignores all of the comments in the IA file. This implies that comments may be placed anywhere in the IA file. The comment section is terminated when the preprocessor finds any of the reserved words used to head another section of the IA file.

Constant Declaration. The constant declaration section is used to declare and equate identifiers with constant values. This optional section is reserved for initializing user-defined constants to be used in the program. The entire section is headed by the reserved word "const", followed by a list of constants assigned to identifiers. Each constant is assigned to an identifier or value with an equal sign. This is shown in the constant declaration section of Figure 4.3. Note the reserved word "const" is required only in the first line of the section. The constant declaration section is concluded when the preprocessor encounters a reserved word used to head one of the other IA file sections.

Type Declaration. The type declaration section is an optional part of the image algebra file which is used to declare user-defined data types. The syntax of the type declaration section is identical to that of the constant declaration section except the reserved word "type" is used in place of "const". See Figure 4.3 for an example. The software switches, iType and tType, are placed here to define

the data type of the image gray levels and template weights, respectively. The default type is real. The type declaration section is terminated when the preprocessor finds a reserved word used to head one of the other IA file sections.

Variable Declaration. The variable declaration section is a required part of the image algebra file. All variable operands used in the program, such as images, templates, and scalars, must be declared (identified) to the compiler in this section. The variable declaration section is headed by the reserved word "var" on the first line followed by one or more lines listing one or more variables separated by commas. Each line of variables is terminated with a colon and a data type. The data types may be either user defined types from the type declaration section, types defined by the image algebra such as image or template, or they may be standard PASCAL data types like real, integer, or character. Figure 4.3 shows some examples of variable declarations for both IA defined and standard PASCAL data types. The variable declaration section is concluded when the preprocessor finds a reserved word used to head one of the other IA file sections.

Program Code. The program code section is the last part of the image algebra file. This is where the image algebra operands are manipulated by the image algebra operators and the flow of program execution is controlled. The program code section is headed by the reserved word "begin" and terminated

with the reserved word "end." In between these two words can be as many image algebra and program control statements as necessary to accomplish the image processing task.

The syntax of the image algebra expressions is relatively straightforward. Any expression may be entered into the IA file as it is written on paper using the AFITIA symbols shown in Figure 4.2. Few restrictions are placed on program lines containing image algebra expressions. Each line may contain multiple image algebra operations. The preprocessor is set up to interpret each line with a left-to-right hierarchy within parentheses; that is, the order of operation is determined from left to right within any level of parentheses. One restriction is that all parentheses be written using the "{" and "}" symbols. Examples of image algebra expressions with multiple operations per line can be found in the sample image algebra source file shown in Figure 4.3.

The initialization of template operands is a simple and flexible process. A template definition is headed by the reserved words "invariant template" followed on the same line by the name of the template. The next line contains the reserved word "begin" to denote the start of the definition. The configuration and weights are initialized on successive lines as pairs of row and column offsets with a weight assigned to each offset pair. The syntax of each line is "(r+x,c+y)=z" where x and y are integer values and z is an

integer or real value depending upon the data type of the template weights. Template definitions are terminated with the reserved word "end". Figure 4.3 shows the definition of two templates, S and T.

Some additional syntax guidelines are carried over from the PASCAL language. These rules affect program execution because of the way the PASCAL compiler interprets program control statements. Basically, PASCAL requires that each group of logically related expressions be delimited with a "begin" and "end" statement. Logically related expressions are those lines of program code under the control of the same program statement such as a loop or conditional branch. An example of this syntax requirement can be found in the "for" loop at the end of Figure 4.3.

Program control statements may not contain image algebra operators but they may contain explicit IA procedure calls. The preprocessor is designed not to interpret any program lines containing reserved words or standard identifiers from the PASCAL language. Therefore, any image algebra operators on the same line as a reserved PASCAL word or PASCAL standard identifier are passed directly to the TRANSLAT.PAS file and may result in errors when they are compiled. However, any AFITIA procedure calls in a program control statement can be compiled successfully.

A list of reserved words and standard identifiers from both the image algebra and PASCAL languages is shown in Figure 4.4. The reserved words are primitive components of both the image algebra and PASCAL which cannot be used as identifiers in an IA program. On the other hand, the PASCAL standard identifiers listed in Figure 4.4 may be redefined in an IA program, but the predefined facility of that identifier is then lost. (3:37-38) Image algebra standard identifiers should be treated as reserved words.

Image and template input/output operations are handled through PASCAL procedure calls. The input/output routines supported by the AFITIA are shown in Figure 4.5. Both read operations prompt the user for the name of the file from which to read. Both write operations write the image to a file named by the user. If the name of the file in the procedure call is left blank, '', the user is prompted for the file name during execution. Appendix C contains the source code for the AFITIA input/output procedures.

The AFITIA contains some extensions to the UFIA set of operators to ease the programming of algorithms in the AFIT image algebra language. In addition to the fourteen elemental operators outlined in Section II, the AFITIA includes three functions and one procedure for image operands, and four procedures for template operands. These operations are shown in Figure 4.5.

VAX PASCAL Reserved Words

and	array	begin	case
const	div	do	downto
else	end	external	file
for	forward	function	goto
if	in	label	mod
nil	not	of	or
packed	procedure	program	record
repeat	set	then	to
type	until	var	varying
while	with		

VAX PASCAL Standard Identifiers

arctan	boolean	byte	char
chr	close	cos	eof
eoln	exp	false	index
input	integer	length	ln
odd	ord	output	pred
read	readln	readvar	reset
rewrite	round	sin	sqr
sqrt	succ	text	true
trunc	write	writeln	writevar

Image Algebra Reserved Words and Standard Identifiers

ConfigTempHConst	ConfigTempVConst	ConfigTempMooreConst
ConfigTempVNConst	ConstImage	dotval
GetImage	GetTemplate	image
ImageEqual	ImageType	invariant
iR1	iR2	iR3
iR4	iR5	iType
MaxImageColumn	MaxImageRow	MaxTempCell
maxval	MaxValImage	minval
MinValImage	PutImage	PutTemplate
template	TemplateType	tR1
tR2	tR3	tR4
tR5	tType	v

Figure 4.4: Reserved Words and Standard Identifiers

Input / Output Operations

operation	AFITIA notation
read image A from the external file 'name'	GetImage (A,'name')
write image A to the external file 'name'	PutImage (A,'name')
read template R from the external file 'name'	GetTemplate (R,'name')
write template R to the external file 'name'	PutTemplate (R,'name')

Image Operations

operation	AFITIA notation
is image A = image B ?	ImageEqual (A,B)
minimum gray value of image A	MinValImage (A)
maximum gray value of image A	MaxValImage (A)
initialize image A to dimension (row x col) with constant pixel gray value of x	ConstImage (A,row,col,x)

Template Operations

operation	AFITIA notation
configure R as a horizontal template of dimension (1 x col) with constant pixel weights of x	ConfigTempHConst (R,col,x)
configure R as a vertical template of dimension (row x 1) with constant pixel weights of x	ConfigTempVConst (R,row,x)
configure R as a Moore template of dimension (row x col) with constant pixel weights of x	ConfigTempMooreConst (R, row,col,x)
configure R as a Von Neumann template of radius (rad) with constant pixel weights of x	ConfigTempVNConst(R,rad,x)

Figure 4.5: Additional AFIT Image Algebra Operations

Building an Executable Module

Once the image processing routine is properly described in the image algebra language, it can be transformed into an executable module with ease by invoking the VAX command file TIA.COM followed by the name of IA file to be transformed. This command file automatically executes the preprocessor to translate the IA file into a PASCAL source code file, executes the PASCAL compiler to transform the source code into an object file, links the object file with the image algebra and I/O routines contained in IAOPER.OBJ and IO.OBJ to produce an executable module, and, finally, renames the executable module with the name of the original IA file. TIA.COM is listed in the appendix.

Figure 4.6 shows the result of preprocessing the sample program from Figure 4.3. After compiling and linking this source code, it becomes a self-contained executable module. Execution of the image processing task is accomplished by running the executable module with the VAX command "RUN filename".

```
[inherit('iiaoper.env','iio.env')]
program image_and_template_operations (input,output);
 const maxloop = 3;
 var r,s,t : template;
     a,b,c : image;
     j : integer;
 begin
  reset (input);
  rewrite (output);
  s.fig(.1.).r:= 0;  s.fig(.1.).c:= 0;  s.fig(.1.).w:=4;
  s.fig(.2.).r:= 0;  s.fig(.2.).c:=+1;  s.fig(.2.).w:=5;
  s.num := 2;
  t.fig(.1.).r:= 0;  t.fig(.1.).c:=-1;  t.fig(.1.).w:=1;
  t.fig(.2.).r:= 0;  t.fig(.2.).c:= 0;  t.fig(.2.).w:=2;
  t.fig(.3.).r:=+1;  t.fig(.3.).c:= 0;  t.fig(.3.).w:=3;
  t.num := 3;
  (***** r:=s(+)t *****)
  TempCirclePlus (s,t,tR1);
  r := tR1;
  getimage (a,'');
  for j := 1 to maxloop do
   begin
   (***** b:=a>=j *****)
   maxval := MaxValImage (a);
   CharImage (a,j,maxval,iR1);
   b := iR1;
   (***** c:=(c+b)[v]r *****)
   ImageAdd (c,b,iR1);
   ImageTempSquareMax (iR1,r,iR2);
   c := iR2;
   end;
  putimage (c,'');
  end.
```

Figure 4.6: PASCAL Source Code After Preprocessing the Sample Program of Figure 4.3

V. Image Algebra Algorithms

To demonstrate the power of the image algebra language and the simplicity of the AFITIA development tool, this section develops three image processing algorithms. The first algorithm, a mean filter, demonstrates the image algebra's power for describing linear image processing tasks. The other two algorithms, a median filter and a local mode filter, exhibit the image algebra's power for representing nonlinear image processing tasks.

A comparison of the two implementations, one in the image algebra and one in PASCAL, of each of the following algorithms demonstrates the tremendous potential productivity increases by a programmer using the image algebra. Due to the concise notation and powerful operators, an image processing engineer can produce reliable image transformation tools with minimal development time. Further, these comparisons highlight the ease of writing image processing algorithms with the image algebra language. They also indicate some areas where the AFITIA needs improvement.

Mean Filter

The mean filter is a linear operation used for noise suppression in images. The filter modifies each image pixel to reflect the mean gray value within a small neighborhood of each image pixel where the neighborhood is defined by the

```
program Mean_Filter (input,output);
 (* Let image A be a two dimensional, integer valued array  *)
 (* of dimensions Image_Rows by Image_Columns.              *)
 (* Let filter F be a 3x3 array of constant weight = 1.     *)
 (* The result of the filter operation is placed in image R *)
 type Image = array (.1..256,1..256.) of integer;
      Template = array (.1..3,1..3.) of integer;
 var  A,R : Image;
      F : Template;
      Image_Rows,Image_Columns,i,j,k,l,sum : integer;
 %Include 'IO_PAS.PAS'
 begin (* program Mean_Filter *)
  reset (input);
  rewrite (output);
  (* read image A from an external file *)
  GetImage (A,'input.img',Image_Rows,Image_Columns);
  (* initialize the filter weights to 1 *)
  for i := 1 to 3 do
   for j := 1 to 3 do F(.i,j.) := 1;
  (* for each pixel in image A *)
  for i := 1 to Image_Rows do
   for j := 1 to Image_Columns do begin
    (* sum the gray values in the neighborhood *)
    sum := 0;
    for k := -1 to 1 do
     for l := -1 to 1 do
      if (i+k>=1) and (i+k<=Image_Rows) and
         (j+l>=1) and (j+l<=Image_Columns)
       then sum := sum + A(.i+k,j+l.)*F(.k+2,l+2.);
    (* divide the sum by the number of neighborhood pixels *)
    (* and place this value into the resulting image       *)
    R(.i,j.) := round (sum/9);
    end; (* for j *)
  (* write the resulting image to OUTPUT.IMG *)
  PutImage (R,'output.img',Image_Rows,Image_Columns);
  end. (* program Mean_Filter *)
```

Figure 5.1: PASCAL Implementation of a Mean Filter

configuration of the filter. This filter has the desirable effect of reducing image noise, but it introduces blurring at step or ramp edges (6:330-331).

Figure 5.1 shows a PASCAL implementation of the mean filter, including the necessary input/output operations to

make it useful as an executable module. The algorithm for this filter is simple. Following the declaration sections, the routine reads an image from a user-specified external file. The external file is assumed to be headed by the row and column dimensions of the image followed by the image data arranged in a two dimensional (row x column) array. After obtaining the input image, the program steps through each pixel of the image. At each pixel of the image, designated by A(i,j), all of the pixel gray values lying under the filter configuration centered over A(i,j) are summed and divided by the number of pixels in the filter configuration. This quotient replaces the gray value at pixel A(i,j).

Figure 5.2 shows the mean filter algorithm described in the ADITIA language. The operation of this algorithm is subtle compared to the direct PASCAL implementation. The declaration and initialization sections are similar. After reading the image from an external file, this algorithm replaces each pixel gray value with the sum of pixel gray values in the neighborhood of each image pixel. Once this operation is completed by the circle-plus operator, a simple scalar division is needed to complete the averaging process: divide the image by the number of pixels in the neighborhood.

A comparison of the Figures 5.1 and 5.2 demonstrates the advantages of the image algebra. As shown, the image algebra routine contains 5 lines of executable (non-comment) code

```
Mean_Filter
 (* Let image A be a two dimensional, integer valued array  *)
 (* of dimensions Image_Rows by Image_Columns.              *)
 (* Let filter F be a 3x3 array of constant weight = 1.     *)
 (* The result of the filter operation is placed in image A *)
 type itype = integer
      ttype = integer
 var  A : image
      F : template
 begin
  (* read image A from an external file *)
  GetImage (A,'input.img')
  (* set the filter weights to 1 *)
  ConfigTempMooreConst (F,3,3,1)
  (* sum the gray values in the neighborhood *)
  A := A (+) F
  (* divide the sum by the number of neighborhood pixels *)
  (* and place this value into the resulting image       *)
  A := A / 9
  (* write the resulting image to OUTPUT.IMG *)
  PutImage (A,'output.img')
  end.
```

Figure 5.2: Image Algebra Implementation of a Mean Filter

compared to 16 lines of code in the PASCAL routine. That is 68% less coding required to implement the same routine in the image algebra. Further, the object code of the IA program requires only 1.5K bytes of storage compared to the 2.5K bytes of storage required by the object code of the direct PASCAL implementation. Additionally, if the user desires to alter the size of the neighborhood of each pixel, the PASCAL program requires recoding of both the filter definition and the limits of program iteration, but the IA program requires only a change of the dimensions used in the template definition.

Subsequent inspection of the executable code size and the execution time of these two programs shows the disadvantages

of this version of the image algebra. The executable code size of the IA program needs 18.5K bytes of storage compared to only 3K bytes of storage for the PASCAL program. The execution time on a 256x256 image is 103.41 seconds for the IA program and 92.52 seconds for the PASCAL program.

Median Filter

The median filter is a nonlinear operation used for noise suppression in images. The filter modifies each image pixel to reflect the median gray value within a small neighborhood of each image pixel where the neighborhood is defined by the configuration of the filter. This filter reduces the noise in an image, and it usually does not affect step or ramp edges (6:330-331).

Figure 5.3 shows a PASCAL implementation of the median filter with the input/output routines necessary to make the program into an executable module. The algorithm for this filter is straightforward. Following the declaration sections, the routine reads an image from a user-specified external described in the mean filter above. After obtaining the input image, the program steps through each pixel of the image. At each pixel of the image, designated by A(i,j), all of the pixel gray values lying under the filter configuration centered over A(i,j) are put into a histogram. The program searches for the median entry in the histogram and places the gray value at this location into the image at pixel A(i,j).

```
program Median_Filter (input,output);
 (*    Let image A be a two dimensional, integer valued     *)
 (* array of dimensions Image_Rows by Image_Columns.        *)
 (*    Let filter F be a 3x3 array of constant weight = 1.  *)
 (*    Let H contain the histogram of the neighborhood      *)
 (* assuming a maximum of 32 gray levels (0-31).            *)
 (*    The result of the filter operation is image R.       *)
 type Image = array (.1..256,1..256.) of integer;
      Template = array (.1..3,1..3.) of integer;
 var  A,R : Image;
      F : Template;
      H : array (.0..31.) of integer:          (* histogram *)
      Image_Rows,Image_Columns,i,j,k,l : integer;
      number,med_val : integer;
 %Include 'IO_PAS.PAS'
 begin (* program Median_Filter *)
  reset (input);
  rewrite (output);
  (* read image A from an external file *)
  GetImage (A,'input.img',Image_Rows,Image_Columns);
  (* set the filter weights to 1 *)
  for i := 1 to 3 do
   for j := 1 to 3 do F(.i,j.) := 1;
  (* for each pixel in image A *)
  for i := 1 to Image_Rows do
   for j := 1 to Image_Columns do begin
    (* build a histogram of the neighborhood *)
    for k := 0 to 31 do H(.k.) := 0;
    for k := -1 to 1 do
     for l := -1 to 1 do
      if (i+k>=1) and (i+k<=Image_Rows) and
         (j+l>=1) and (j+l<=Image_Columns)
       then H(.A(.i+k,j+l.).) := H(.A(.i+k,j+l.).) + 1;
    (* find the number of non-zero histogram values *)
    number := 0;
    for k := 1 to 31 do
     if H(.k.) <> 0 then number := number + 1;
    (* the median gray value is the median entry in H *)
    med_val := round (number/2);
    (* find and place the median gray value  nto image R *)
    number := 0;
    for k := 1 to 31 do begin
     if H(.k.) <> 0 then number := number + 1;
     if number = med_val then R(.i,j.) := H(.k.);
     end; (* for k *)
    end; (* for j *)
  (* write the resulting image to an external file *)
  PutImage (R,'output.img',Image_Rows,Image_Columns);
  end. (* program Median_Filter *)
```

Figure 5.3: PASCAL Implementation of a Median Filter

```
Median_Filter
 (* Let image A be a two dimensional, integer valued array *)
 (* of dimensions Image_Rows by Image_Columns.              *)
 (* Let filter F be a 3x3 array of constant weight = 1.     *)
 (* The result of the filter operation is in image R.       *)
 var i1,A,B,C,R : image
     F : template
     x,dot : integer
 begin
  (* read image A from an external file *)
  GetImage (A,'input.img')
  (* initialize F=1 *)
  ConfigTempMooreConst (F,3,3,1)
  (* initialize i1=1, R=0 the same size as A *)
  ConstImage (i1,A.row,A.col,1)
  ConstImage (R,A.row,A.col,0)
  (* calculate the median frequency threshold *)
  C := (i1 (+) F) / 2
  for x := MinValImage (A) to MaxValImage (A) do begin
   dot := (A=x).i1
   if dot > 0 then begin
    (* calculate the frequency of image pixels *)
    (* under template F with gray value >= x   *)
    B := (A>=x) (+) F
    (* update the pixel gray value if frequency is *)
    (* greater than the median frequency threshold *)
    R := R v ((B-C)>0)*x)
    end (* if dot then *)
   end (* for x do*)
  (* write the resulting image to an external image *)
  PutImage (R,'output.img')
  end.
```

Figure 5.4: Image Algebra Implementation of a Median Filter

Figure 5.4 shows the median filter algorithm written in the image algebra language. This operation of this algorithm is not as obvious as the direct PASCAL implementation. After the usual declarations and obtaining the input image, A, this algorithm initializes three more images of identical size to A. Image i1, which is used by the dot product operation to count pixels, is set to one at every pixel. Image C is set to

the median frequency threshold at each pixel. This means each pixel in C assumes a value corresponding to the median number of pixels within the neighborhood of that image pixel. Image R, the result image, is initialized to zero at every pixel.

The remainder of the algorithm is iterated by gray level from the minimum gray value to the maximum gray value in image A. During each iteration, image B is set to the frequency of gray levels within the neighborhood of each pixel greater than the present gray level of the iteration. Then, each pixel that has a frequency greater than the median frequency threshold is updated with the present gray level of the iteration. Thus, each pixel is updated with the a new gray value until its frequency is greater than or equal to the median frequency threshold.

A comparison of the Figures 5.3 and 5.4 demonstrates the advantages of the image algebra description. As shown, the image algebra version contains 13 lines of executable (non-comment) code compared to 24 lines in the PASCAL version. The result is better than 45% less programming required by the image algebra implementation. Additionally, the object code of the IA program requires only 2.5K bytes of storage compared to the 3K bytes of storage required by the object code of the direct PASCAL implementation. As in the IA implementation of the mean filter, the alteration of the pixel neighborhood in

the IA version of the median filter requires only a change in the dimensions of the template definition.

Further inspection of the executable code size and the execution time of these two programs shows the disadvantages of this version of the image algebra. The executable code size of the IA program requires 19K bytes of storage compared to only 3K bytes of storage required by the PASCAL program. The execution time on a 256x256 image is 231.36 seconds for the IA program and 167.16 seconds for the PASCAL program.

Local Mode Filter

The local mode filter is a nonlinear operation useful for noise suppression in images also. The filter modifies each image pixel to reflect the most frequent (mode) gray value within a small neighborhood of each image pixel where the neighborhood is defined by the configuration of the filter. This filter can remove noise from an image without introducing errors in step or ramp edges. Furthermore, it is usually more responsive to local image context than the median filter. For example, assume the image depicted on the next page in Figure 5.5 is to be filtered and the image pixel under modification is in the center. Without additional knowledge of the image gray values, the logical choice for the new gray value would be either one or seven. The local mode filter will chose one of those values. However, the median filter will chose three, essentially ignoring the context of the image neighborhood.

1	1	7
1	2	7
3	7	7

Figure 5.5: Image for Local Mode Filter Example

Figure 5.6 shows a PASCAL implementation of the local mode filter with the input/output routines necessary to make the program into an executable module. The algorithm for this filter is nearly identical to the median filter. The only difference is that the image pixel gray value is updated with the mode, rather than the median, of the histogram.

Figure 5.7 shows the same algorithm written in the image algebra. This implementation is similar to the image algebra description of the median filter. After the declarations and obtaining the input image, A, this algorithm initializes three images to zero: B, which is used to hold the frequency of pixels within each neighborhood at a gray level; C, which is used to hold the maximum frequency of pixels within each neighborhood at a gray level; and R, which is used as the resultant image. Image C is set to the median frequency threshold at each pixel.

The remainder of the algorithm is iterated by gray level from the minimum gray value to the maximum gray value in image A. During each iteration, image B is set to the frequency of gray levels within the neighborhood of each pixel

```
program Local_Mode_Filter (input,output);
 (*   Let image A be a two dimensional, integer valued     *)
 (* array of dimensions Image_Rows by Image_Columns.       *)
 (*   Let filter F be a 3x3 array of constant weight = 1.  *)
 (*   Let H contain the histogram of the neighborhood      *)
 (* assuming a maximum of 32 gray levels (0-31).           *)
 (*   The result of the filter operation is image R.       *)
 type Image = array (.1..256,1..256.) of integer;
      Template = array (.1..3,1..3.) of integer;
 var  A,R : Image;
      F : Template;
      H : array (.0..31.) of integer:        (* histogram *)
      Image_Rows,Image_Columns,i,j,k,l : integer;
      max_val : integer;
 %Include 'IO_PAS.PAS'
 begin (* program Local_Mode_Filter *)
  reset (input);
  rewrite (output);
  (* read image A from an external file *)
  GetImage (A,'input.img',Image_Rows,Image_Columns);
  (* set the filter weights to 1 *)
  for i := 1 to 3 do
   for j := 1 to 3 do F(.i,j.) := 1;
  (* for each pixel in image A *)
  for i := 1 to Image_Rows do
   for j := 1 to Image_Columns do begin
    (* build a histogram of the neighborhood *)
    for k := 0 to 31 do H(.k.) := 0;
    for k := -1 to 1 do
     for l := -1 to 1 do
      if (i+k>=1) and (i+k<=Image_Rows) and
         (j+l>=1) and (j+l<=Image_Columns)
       then H(.A(.i+k,j+l.).) := H(.A(.i+k,j+l.).) + 1;
    (* find the mode of the histogram *)
    max_val := H(.A(.i,j.).);
    for k := 1 to 31 do
     if H(.k.) > max_val then max_val := H(.k.);
    (* find and place the mode's gray value into image R *)
    for k := 1 to 31 do
     if H(.k.) = max_val then R(.i,j.) := H(.k.);
    end; (* for j *)
  (* write the resulting image an external file *)
  PutImage (R,'output.img',Image_Rows,Image_Columns);
  end. (* program Local_Mode_Filter *)
```

Figure 5.6: PASCAL Implementation of a Local Mode Filter

```
Local_Mode_Filter
 (* Let image A be a two dimensional, integer valued     *)
 (* array of dimensions Image_Rows by Image_Columns.     *)
 (* Let filter F be a 3x3 array of constant weight = 1.  *)
 (* The result of the filter operation is image R.       *)
 var A,B,C,R : image
     F : template
     x,dot : integer
 begin
  (* read image A from an external file *)
  GetImage (A,'input.img')
  (* initialize the filter F=1 *)
  ConfigTempMooreConst (F,3,3,1)
  (* initialize B = C = R = 0 the same size as A *)
  ConstImage (R,A.row,A.col,0)
  B := C := R
  for x := MinValImage (A) to MaxValImage (A) do begin
   B := (A=x) (+) R
   R := R v {{(BvC-C)>0}*x}
   C := B v C
   end (* for x *)
  (* write the resulting image to an external file *)
  PutImage (R,'output.img')
  end.
```

Figure 5.7: Image Algebra Implementation of a Local Mode Filter

equal to the present gray level of the iteration. Then, each pixel having a frequency greater than the present maximum frequency of that pixel, held by C, is updated with the present gray level of the iteration. Thus, each pixel is updated with the a new gray value as long as its frequency is greater than the present maximum frequency for that pixel. The final step of each iteration updates the maximum frequency at each pixel.

Once again, a comparison of the two implementations shows the programming advantages of the image algebra. As shown, the image algebra version contains 10 lines of executable

(non-comment) code compared to 20 lines in the PASCAL version. This translates to 50% less programming required by the image algebra implementation. Further, the object code of the IA program needs 2K bytes of storage compared to the 3K bytes of storage required by the object code of the direct PASCAL implementation. Like the other two algorithms, changing the size and configuration of the neighborhood is easier with the IA version, too.

Inspection of the executable code size and the execution time of these two programs shows the disadvantages of this version of the image algebra. The executable code size of the IA program requires 18.5K bytes of storage compared to only 3K bytes of storage needed by the PASCAL program. The execution time on a 256x256 image is 203.82 seconds for the IA program and 132.22 seconds for the PASCAL program.

Image Algebra vs High Level Language Routines

As can be seen by the previous examples, the power and simplicity of the image algebra is tremendous. The image algebra provides a concise notation for algorithm development that removes the engineer's attention to details required when programming in PASCAL or some other high level language. This notation allows the image processing engineer to concentrate on building image processing tools rather than on writing and debugging computer programs.

		code size (Kbytes)			execution time (seconds)		
P A S C A L	program	source	object	execute	16x16	64x64	256x256
	mean	1.5	2.5	3.0	2.03	7.44	92.52
	median	2.5	3.0	3.0	2.38	12.14	167.16
	mode	2.0	3.0	3.0	2.25	9.95	132.22
I A	mean	1.0	1.5	18.5	5.85	11.63	103.41
	median	1.5	2.5	19.0	30.12	41.52	231.36
	mode	1.0	2.0	18.5	29.41	39.39	203.82

Table 5.1: Code Size and Execution Time for PASCAL and Image Algebra Example Algorithms

Futhermore, accommodating arbitrary neighborhood sizes and configurations is much easier to program into the image algebra: merely alter the configuration of the templates.

Even though the IA provides increased programming productivity for image processing routines, the size of the executable code and slower execution times are significant detriments to this version of the AFITIA. Table 5.1 shows a summary of the storage requirements and execution times for the two versions of each image processing algorithm. The top set of data corresponds to the PASCAL implementation of each algorithm, and the bottom set of data corresponds to the image algebra implementation. The table shows that both the source code and object code sizes are smaller in the IA programs, but the executable code size is much larger. This is due to the linking of the entire library of IA operations to each IA module whether or not each operation is needed.

The execution times on 16x16, 64x64, and 256x256 binary images show that the IA implementations are much slower than the direct PASCAL implementations. The execution times of the PASCAL implementations display the expected n^2 increases in execution time where n is the image dimension. Even though Table 5.1 does not display n^2 increases in execution time for the IA programs, this does not imply that the IA algorithms are more efficient for larger images. The increased execution time due to the larger images is hidden by the high execution overhead from all of the procedure and function calls.

The AFIT image algebra operations and preprocessor should be modified to reduce both of these problems. One method of reducing the code size is to program the preprocessor so that only those basic IA operations used by the program are linked to the object code. Reducing the execution time of the IA programs will be more difficult due the overhead involved in procedure and function calls.

As was shown in the previous examples, the implementation of an image processing algorithm in the image algebra can be very different than its direct implementation in PASCAL. The algorithm must be formulated differently for the two languages because of the difference in elemental operations. Because the IA forces an image processing engineer to approach the programming from a new and very different viewpoint, it is difficult to determine which method is easier to implement at

this point. However, once an engineer becomes familiar with the IA operations, it is anticipated the IA formulation will become easier and its implementation will be preferred.

VI. Observations and Recommendations

The image processing environment designed and implemented in this project is a useful and highly flexible development tool. Through the use of a preprocessing program, it is capable of translating image processing algorithms, written with image algebra operators and PASCAL control structures, into executable programs. The entire process can be automated through the use of a VAX command file.

The AFIT image algebra language and preprocessor allow the construction of image processing algorithms with minimal development time and programming effort. The actual program code can be reduced by 45-70%, and the low level programming details of each algorithm are managed by the IA language and preprocessor. The AFITIA supports real and integer valued images and templates, and the syntax of the IA source file is relatively simple and flexible. The AFITIA support of all PASCAL control structures provides an IA language capable of performing any image-to-image transformation.

One major problem with this design appeared and remains unresolved: complete software independence. The dependencies of the preprocessor target language have not been removed from the AFITIA language. Since the ultimate IA language will be translated to many different computer languages, the image algebra language must define and support program control

statements common to the more popular high order languages. Further research will provide valuable guidance about which statements should be supported and the final form of the image algebra language.

Some other shortcomings of this version of the AFITIA are a lack of support for complex images and templates, the large size of the executable code for eacn image algebra algorithm, and their slower execution. The correction of these problems through a more intelligent linking procedure and faster code for the IA operations is highly recommended.

Another topic of interest stemming from this project is the inherent parallelism of the image algebra operators. Subsequent research could determine which operations are amenable to parallel execution in either vector (1-D array) or image (2-D array) processing architectures.

Overall, the blending of the image algebra developed at the University of Florida with the AFIT preprocessor makes significant progress toward a simple, powerful, and universal image processing tool.

Appendix A: AFIT Image Algebra Operations

```
[environment('riaoper.env')]
Module Real_IA_Operations;

(*****************************************************************************)
(* This file contains all of the basic image and template operations from  *)
(* the Image Algebra developed for the AFATL, Eglin AFB, FL.  The routines *)
(* in this file are written for real valued images and templates.  A       *)
(* number of AFIT extensions are included.                                 *)
(*****************************************************************************)

(* global constants, types, and variables *)
const MaxImageRow    =  22;          (* maximum image row dimension       *)
      MaxImageColumn =  32;          (* maximum image column dimension    *)
      MaxTempCell    = 100;          (* maximum number of template pixels *)
      NameLen        =  80;          (* maximum length of file names      *)
type  ImageType      =  real;        (* data type of image gray values    *)
      TemplateType   =  real;        (* data type of template weights     *)
      Image = record                 (* PASCAL implementation of image operand *)
                row  : integer;      (* image row dimension *)
                col  : integer;      (* image column dimension *)
                gray : array (.1..MaxImageRow,1..MaxImageColumn.) of ImageType;
                end;                 (* image record *)
      Template = record              (* PASCAL implementation of template operand  *)
                num : integer;       (* number of pixels within the configuration *)
                cfg : array (.1..MaxTempCell.) of record   (* configuration *)
                        r : integer;       (* row offset of pixel     *)
                        c : integer;       (* column offset of pixel  *)
                        w : TemplateType;  (* weight of pixel         *)
                        end;         (* array record *)
                end;                 (* template record *)
      OperStr = varying[3] of char;
      NameStr = varying[NameLen] of char;
var   iR1,iR2,iR3,iR4,iR5  : Image;      (*  intermediate image operands   *)
      tR1,tR2,tR3,tR4,tR5  : Template;   (*  intermediate template operands *)
      dotval,minval,maxval : ImageType;  (*  intermediate scalar operands   *)
```

```
(*****************************************************************************)
(*                    UNARY and BINARY IMAGE OPERATIONS                      *)
(*****************************************************************************)

function ImageEqual ( A,B : Image) : boolean;
 (* return true if A = B, pointwise *)
 var i,j : integer;
     equal : boolean;
 begin (* function ImageEqual *)
  if (A.row=B.row) and (A.col=B.col) then equal := true else equal := false;
  i := 1;
  while (equal = true) and (i <= A.row) do begin
   j :=1;
   while (equal = true) and (j <= A.col) do begin
    if A.gray(.i,j.) <> B.gray(.i,j.) then equal := false;
    j := j + 1;
    end; (* while j *)
   i := i + 1;
   end; (* while i *)
  ImageEqual := equal;
  end; (* function ImageEqual *)

procedure ConstImage ( var A : Image; row,col : integer; value : ImageType);
 (* initialize image A to a constant value: A = value *)
 var i,j : integer;
 begin (* procedure ConstImage *)
  A.row := row; A.col := col;
  for i := 1 to row do
   for j := 1 to col do A.gray(.i,j.) := value;
  end; (* procedure ConstImage *)

function MaxValImage (A : Image): ImageType;
 (* return the maximum pixel value in the image *)
 var i,j : integer;
     maxval : ImageType;
 begin (* procedure MaxValImage *)
  maxval := A.gray(.1,1.);
  for i := 1 to A.row do
   for j := 1 to A.col do
    if A.gray(.i,j.) > maxval then maxval := A.gray(.i,j.);
  MaxValImage := maxval;
  end; (* procedure MaxValImage *)
```

```
function MinValImage (A : Image) : ImageType;
 (* returns the minimum pixel value in the image *)
 var i,j : integer;
     minval : ImageType;
 begin (* procedure MinValImage *)
  minval := A.gray(.1,1.);
  for i := 1 to A.row do
   for j := 1 to A.col do
    if A.gray(.i,j.) < minval then minval := A.gray(.i,j.);
  MinValImage := minval;
  end; (* procedure MinValImage *)

procedure AbsValImage ( A : Image; var C : Image);
 (* absolute value of the image: C = |A| *)
 var i,j : integer;
 begin (* procedure AbsValImage *)
  C.row := A.row; C.col := A.col;
  for i := 1 to A.row do
   for j := 1 to A.col do C.gray(.i,j.) := abs (A.gray(.i,j.));
  end; (* procedure AbsValImage *)

procedure ImageAdd ( A,B : Image; var C : Image);
 (* pointwise image addition: C = A + B *)
 var i,j : integer;
 begin (* procedure ImageAdd *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then C.gray(.i,j.) := A.gray(.i,j.) + B.gray(.i,j.);
    if (i > A.row) or (j > A.col) then C.gray(.i,j.) := B.gray(.i,j.);
    if (i > B.row) or (j > B.col) then C.gray(.i,j.) := A.gray(.i,j.);
    end; (* for j do *)
  end; (* procedure ImageAdd *)

procedure ImageSubtract ( A,B : Image; var C : Image);
 (* pointwise image subtraction, C = A - B *)
 var i,j : integer;
 begin (* procedure ImageSubtract *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then C.gray(.i,j.) := A.gray(.i,j.) - B.gray(.i,j.);
    if (i > A.row) or (j > A.col) then C.gray(.i,j.) := -B.gray(.i,j.);
    if (i > B.row) or (j > B.col) then C.gray(.i,j.) := A.gray(.i,j.);
    end; (* for j do *)
  end; (* procedure ImageSubtract *)
```

```
procedure ImageMultiply ( A,B : Image; var C : Image);
 (* pointwise image multiplication: C = A x B *)
 var i,j : integer;
 begin (* procedure ImageMultiply *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then C.gray(.i,j.) := A.gray(.i,j.) * B.gray(.i,j.)
     else C.gray(.i,j.) := 0;
    end; (* for j do *)
  end; (* procedure ImageMultiply *)

procedure ImageDivide ( A,B : Image; var C : Image);
 (* pointwise image division: C = A / B *)
 var i,j : integer;
 begin (* procedure ImageDivide *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and
       (j <= B.col) and (B.gray(.i,j.) <> 0)
     then C.gray(.i,j.) := A.gray(.i,j.) / B.gray(.i,j.)
     else C.gray(.i,j.) := 0;
  end; (* procedure ImageDivide *)

procedure ImageExponent (A,B : Image; var C : Image);
 (* pointwise exponentiation of an image by an image: C := A**B = exp(B*ln(A)) *)
 var i,j,k : integer;
 begin (* procedure ImageExponent *)
  C.row := A.row; C.col := A.col;
  for i := 1 to C.row do
   for j := 1 to C.col do
    if A.gray(.i,j.) <> 0 then C.gray(.i,j.) := ln(abs(A.gray(.i,j.)))
                          else C.gray(.i,j.) := 0;
  ImageMultiply (C,B,C);
  for i := 1 to C.row do
   for j := 1 to C.col do C.gray(.i,j.) := exp(C.gray(.i,j.));
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (A.gray(.i,j.) < 0) and (C.gray(.i,j.) <> 0)
     then C.gray(.i,j.) := 1 / C.gray(.i,j.);
    if A.gray(.i,j.) = 0 then C.gray(.i,j.) := 0;
    end; (* for j do *)
  end; (* procedure ImageExponent *)
```

```
procedure ImageMax ( A,B : Image; var C : Image);
 (* return the pointwise maximum of images A and B: C = max (A,B) *)
 var i,j : integer;
 begin (* procedure ImageMax *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then if A.gray(.i,j.) >= B.gray(.i,j.) then C.gray(.i,j.) := A.gray(.i,j.)
                                            else C.gray(.i,j.) := B.gray(.i,j.);
    if (i > A.row) or (j > A.col)
     then if B.gray(.i,j.) > 0 then C.gray(.i,j.) := B.gray(.i,j.)
                               else C.gray(.i,j.) := 0;
    if (i > B.row) or (j > B.col)
     then if A.gray(.i,j.) > 0 then C.gray(.i,j.) := A.gray(.i,j.)
                               else C.gray(.i,j.) := 0;
    end; (* for j do *)
  end; (* procedure ImageMax *)

procedure ImageMin ( A,B : Image; var C : Image);
 (* return the pointwise minimum of images A and B: C = min (A,B) *)
 var i,j : integer;
 begin (* procedure ImageMin *)
  if A.row >= B.row then C.row := A.row else C.row := B.row;
  if A.col >= B.col then C.col := A.col else C.col := B.col;
  for i := 1 to C.row do
   for j := 1 to C.col do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then if A.gray(.i,j.) <= B.gray(.i,j.) then C.gray(.i,j.) := A.gray(.i,j.)
                                            else C.gray(.i,j.) := B.gray(.i,j.);
    if (i > A.row) or (j > A.col)
     then if B.gray(.i,j.) < 0 then C.gray(.i,j.) := B.gray(.i,j.)
                               else C.gray(.i,j.) := 0;
    if (i > B.row) or (j > B.col)
     then if A.gray(.i,j.) < 0 then C.gray(.i,j.) := A.gray(.i,j.)
                               else C.gray(.i,j.) := 0;
    end; (* for j do *)
  end; (* procedure ImageMin *)
```

```
procedure CharImage ( A : Image; operator : OperStr; B : Image; var C : Image);
 (* return a binary image with pixels=1 for A [ <,<=,=,>,>= ] B true *)
 (* This procedure does not use image algebra primitives in order to    *)
 (* reduce the memory and computational requirements of the routine     *)
 var i,j : integer;
 begin (* procedure CharImage *)
  C.row := A.row; C.col := A.col;
  if operator = '<' then
   for i := 1 to A.row do
    for j := 1 to A.col do
     if A.gray(.i,j.) < B.gray(.i,j.) then C.gray(.i,j.) := 1
                                      else C.gray(.i,j.) := 0;
  if operator = '<=' then
   for i := 1 to A.row do
    for j := 1 to A.col do
        A.gray(.i,j.) <= B.gray(.i,j.) then C.gray(.i,j.) := 1
                                       else C.gray(.i,j.) := 0;
  if operator = '=' then
   for i := 1 to A.row do
    for j := 1 to A.col do
     if A.gray(.i,j.) =  B.gray(.i,j.) then C.gray(.i,j.) := 1
                                       else C.gray(.i,j.) := 0;
  if operator = '>=' then
   for i := 1 to A.row do
    for j := 1 to A.col do
     if A.gray(.i,j.) >=  B.gray(.i,j.) then C.gray(.i,j.) := 1
                                        else C.gray(.i,j.) := 0;
  if operator = '>' then
   for i := 1 to A.row do
    for j := 1 to A.col do
     if A.gray(.i,j.) >  B.gray(.i,j.) then C.gray(.i,j.) := 1
                                       else C.gray(.i,j.) := 0;
  end; (* procedure CharImage *)

procedure ImageDot ( A,B : Image; var Sum : ImageType);
 (* dot product of two images: result := A (dot) B *)
 var i,j,RowMax,ColMax : integer;
 begin (* procedure ImageDot *)
  if A.row >= B.row then RowMax := A.row else RowMax := B.row;
  if A.col >= B.col then ColMax := A.col else ColMax := B.col;
  Sum := 0;
  for i := 1 to RowMax do
   for j := 1 to ColMax do begin
    if (i <= A.row) and (i <= B.row) and (j <= A.col) and (j <= B.col)
     then Sum := Sum + A.gray(.i,j.) * B.gray(.i,j.);
    end; (* for j do *)
  Result := Sum;
  end; (* procedure ImageDot *)
```

```
(****************************************************************************)
(*                    UNARY and BINARY TEMPLATE OPERATIONS                   *)
(****************************************************************************)

procedure ConfigTempHConst ( var A : Template; Col : integer; value : TemplateType);
 (* initialize all pixels in the 1 x Col horizontal template to value *)
 (* the center is assumed to be the physical center of the configuration *)
 var j,cnt : integer;
 begin (* procedure ConfigTempHConst *)
  (* initialize the template configuration to zero *)
  for j := 1 to MaxTempCell do begin
   A.cfg(.j.).r := 0;
   A.cfg(.j.).c := 0;
   A.cfg(.j.).w := 0;
   end; (* for j do *)
  cnt := 0;
  for j := -(Col div 2) to (Col div 2) do begin
   cnt := cnt + 1;
   A.cfg(.cnt.).r := 0;
   A.cfg(.cnt.).c := j;
   A.cfg(.cnt.).w := value;
   end; (* for j do *)
  A.num := cnt;
  end; (* procedure ConfigTempHConst *)

procedure ConfigTempVConst ( var A : Template; Row : integer; value : TemplateType);
 (* initialize all pixels in the Row x 1 vertical template to value *)
 (* the center is assumed to be the physical center of the configuration *)
 var i,cnt : integer;
 begin (* procedure ConfigTempHConst *)
  (* initialize the template configuration to zero *)
  for i := 1 to MaxTempCell do begin
   A.cfg(.i.).r := 0;
   A.cfg(.i.).c := 0;
   A.cfg(.i.).w := 0;
   end; (* for i do *)
  cnt := 0;
  for i := -(Row div 2) to (Row div 2) do begin
   cnt := cnt + 1;
   A.cfg(.cnt.).r := i;
   A.cfg(.cnt.).c := 0;
   A.cfg(.cnt.).w := value;
   end; (* for j do *)
  A.num := cnt;
  end; (* procedure ConfigTempVConst *)
```

```
procedure ConfigTempMooreConst (var A:Template; Row,Col:integer; value : TemplateType);
 (* initialize all pixels in the Row by Col Moore template to value *)
 (* the center is assumed to be the physical center of the configuration *)
 var i,j,cnt : integer;
 begin (* procedure ConfigTempMooreConst *)
  (* initialize the template configuration to zero *)
  for i := 1 to MaxTempCell do begin
   A.cfg(.i.).r := 0;
   A.cfg(.i.).c := 0;
   A.cfg(.i.).w := 0;
   end; (* for i do *)
  cnt := 0;
  for i := -(Row div 2) to (Row div 2) do
   for j := -(Col div 2) to (Col div 2) do begin
    cnt := cnt + 1;
    A.cfg(.cnt.).r := i;
    A.cfg(.cnt.).c := j;
    A.cfg(.cnt.).w := value;
    end; (* for j do *)
  A.num := cnt;
  end; (* procedure ConfigTempMooreConst *)

procedure ConfigTempVNConst (var A : Template; rad : integer; value : TemplateType);
 (* initialize a Von Neumann template configuration of radius rad to value *)
 (* the center is assumed to be the physical center of the configuration *)
 var i,number : integer;
 begin (* procedure ConfigTempVNConst *)
  (* initialize the template configuration to zero *)
  for i := 1 to MaxTempCell do begin
   A.cfg(.i.).r := 0; A.cfg(.i.).c := 0;
   A.cfg(.i.).w := 0;
   end; (* for i do *)
  number := 0;
  for i := -rad to -1 do begin
   number := number + 1;
   A.cfg(.number.).r := i; A.cfg(.number.).c := 0;
   A.cfg(.number.).w := value;
   end; (* for i do *)
  for i := -rad to rad do begin
   number := number + 1;
   A.cfg(.number.).r := 0; A.cfg(.number.).c := i;
   A.cfg(.number.).w := value;
   end; (* for i do *)
  for i := 1 to rad do begin
   number := number + 1;
   A.cfg(.number.).r := i; A.cfg(.number.).c := 0;
   A.cfg(.number.).w := value;
   end; (* for i do *)
  A.num := number;
  end; (* procedure ConfigTempVNConst *)
```

```
procedure TempAdd ( A,B : Template; var C : Template);
 (* pointwise addition of templates:  C = A + B, for A U B *)
 var cnt,cntA,cntB : integer;
     intersect : boolean;
 begin (* procedure TempAdd *)
  C := A;
  cnt := A.num;
  for cntB := 1 to B.num do begin
   intersect := false;
   for cntA := 1 to A.num do (* add intersecting template pixels *)
    if (A.cfg(.cntA.).r = B.cfg(.cntB.).r) and (A.cfg(.cntA.).c = B.cfg(.cntB.).c) and
       (intersect = false) then begin
     C.cfg(.cntA.).w := A.cfg(.cntA.).w   B.cfg(.cntB.).w;
     intersect := true;
     end; (* if intersect true *)
   if intersect = false then begin (* append nonintersecting pixels *)
    cnt := cnt + 1;
    C.cfg(.cnt.).c := B.cfg(.cntB.).c;
    C.cfg(.cnt.).r := B.cfg(.cntB.).r;
    C.cfg(.cnt.).w := B.cfg(.cntB.).w;
    end; (* if intersect false then *)
   end; (* for cntB do *)
  C.num := cnt;
  end; (* procedure TempAdd *)

procedure TempSubtract ( A,B : Template; var C : Template);
 (* pointwise subtraction of templates:  C = A - B, for A U B *)
 var cnt,cntA,cntB : integer;
     intersect : boolean;
 begin (* procedure TempSubtract *)
  C := A;
  cnt := A.num;
  for cntB := 1 to B.num do begin
   intersect := false;
   for cntA := 1 to A.num do (* subtract intersecting template pixels *)
    if (A.cfg(.cntA.).r   B.cfg(.cntB.).r) and (A.cfg(.cntA.).c = B.cfg(.cntB.).c) and
       (intersect = false) then begin
     C.cfg(.cntA.).w := A.cfg(.cntA.).w - B.cfg(.cntB.).w;
     intersect := true;
     end; (* if intersect true then *)
   if intersect = false then begin (* append nonintersecting pixels *)
    cnt :=  cnt + 1;
    C.cfg(.cnt.).c :=   B.cfg(.cntB.).c;
    C.cfg(.cnt.).r :=   B.cfg(.cntB.).r;
    C.cfg(.cnt.).w := -B.cfg(.cntB.).w;
    end; (* if intersect false then *)
   end; (* for cntB do *)
  C.num :=  cnt,
  end; (* procedure TempSubtract *)
```

```
procedure TempMultiply ( A,B : Template; var C : Template);
 (* pointwise multiplication of templates :  C = A x B, for A ∩ B *)
 var cnt,cntA,cntB : integer;
 begin (* procedure TempMultiply *)
  cnt := 0;
  for cntA := 1 to A.num do
   for cntB := 1 to B.num do (* multiply intersecting template pixels *)
    if (A.cfg(.cntA.).r=B.cfg(.cntB.).r) and (A.cfg(.cntA.).c=B.cfg(.cntB.).c) then begin
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w * B.cfg(.cntB.).w;
     end; (* if *)
  C.num := cnt;
  end; (* procedure TempMultiply *)

procedure TempDivide ( A,B : Template; var C : Template);
 (* pointwise division of templates :  C = A / B, for A ∩ B *)
 var cnt,cntA,cntB : integer;
 begin (* procedure TempDivide *)
  cnt := 0;
  for cntA := 1 to A.num do
   for cntB := 1 to B.num do (* divide intersecting template pixels *)
    if (A.cfg(.cntA.).r=B.cfg(.cntB.).r) and (A.cfg(.cntA.).c=B.cfg(.cntB.).c) then begin
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c;
     if B.cfg(.cntB.).w <> 0
      then C.cfg(.cnt.).w := A.cfg(.cntA.).w / B.cfg(.cntB.).w
      else C.cfg(.cnt.).w := 0;
    end; (* if *)
  C.num := cnt;
  end; (* procedure TempDivide *)

procedure TempMax ( A,B : Template; var C : Template);
 (* pointwise maximum of templates :  C = max (A,B), for A ∪ B *)
 var cnt,cntA,cntB : integer;
     intersect : boolean;
 begin (* procedure TempMax *)
  C := A;
  cnt := A.num;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do begin
   intersect := false;
   for cntA := 1 to A.num do (* maximize intersecting template pixels *)
    if (A.cfg(.cntA.).c = B.cfg(.cntB.).c) and (A.cfg(.cntA.).r = B.cfg(.cntB.).r) and
       (intersect = false) then begin
     if A.cfg(.cntA.).w < B.cfg(.cntB.).w then C.cfg(.cntA.).w := B.cfg(.cntB.).w;
     intersect := true;
     end; (* if intersect true then *)
```

```
   if intersect = false then begin (* append nonintersecting pixels *)
    cnt := cnt + 1;
    C.cfg(.cnt.).r := B.cfg(.cntB.).r;
    C.cfg(.cnt.).c := B.cfg(.cntB.).c;
    if B.cfg(.cntB.).w > 0 then C.cfg(.cnt.).w := A.cfg(.cntB.).w
                           else C.cfg(.cnt.).w := 0;
    end; (* if intersect false then *)
   end; (* for cntB do *)
  C.num := cnt;
  end; (* procedure TempMax *)

procedure TempMin ( A,B : Template; var C : Template);
 (* pointwise minimum of templates :  C = min (A,B), for A U B *)
 var cnt,cntA,cntB : integer;
     intersect : boolean;
 begin (* procedure TempMin *)
  C := A;
  cnt := A.num;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do begin
   intersect := false;
   for cntA := 1 to A.num do (* maximize intersecting template pixels *)
    if (A.cfg(.cntA.).c=B.cfg(.cntB.).c) and (A.cfg(.cntA.).r=B.cfg(.cntB.).r) and
       (intersect = false) then begin
     if A.cfg(.cntA.).w > B.cfg(.cntB.).w then C.cfg(.cntA.).w := B.cfg(.cntB.).w;
     intersect := true;
     end; (* if intersect true then *)
   if intersect = false then begin (* append nonintersecting pixels *)
    cnt := cnt + 1;
    C.cfg(.cnt.).r := B.cfg(.cntB.).r;
    C.cfg(.cnt.).c := B.cfg(.cntB.).c;
    if B.cfg(.cntB.).w < 0 then C.cfg(.cnt.).w := B.cfg(.cntB.).w
                           else C.cfg(.cnt.).w := 0;
    end; (* if intersect false then *)
   end; (* for cntB do *)
  C.num := cnt;
  end; (* procedure TempMin *)

procedure TempCirclePlus ( A,B : Template; var C : Template);
 var i,cnt,cntA,cntB : integer;
     duplicate : boolean;
 begin (* procedure TempCirclePlus *)
  cnt := 0;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do
   (* compute B(bx,by) * A(ax,ay) for template A centered *)
   (* on B(bx,by) and sum the product into C(ax+bx,ay+by) *)
   for cntA := 1 to A.num do begin
    duplicate := false;
```

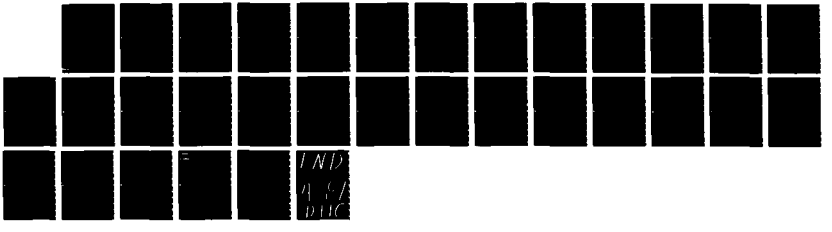
```
    (* find the pixel in the template configuration if it exists *)
    for i := 1 to cnt do
     if (C.cfg(.i.).r = A.cfg(.cntA.).r + B.cfg(.cntB.).r) and
        (C.cfg(.i.).c = A.cfg(.cntA.).c + B.cfg(.cntB.).c)
      then begin (* pixel already in template C configuration *)
       C.cfg(.i.).w := C.cfg(.i.).w + A.cfg(.cntA.).w * B.cfg(.cntB.).w;
       duplicate := true;
       end; (* if pixel found *)
    if duplicate = false then begin (* append nonintersecting pixels *)
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w * B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cnt A do *)
  C.num := cnt;
  end; (* procedure TempCirclePlus *)

procedure TempCircleMax ( A,B : Template; var C : Template);
 var i,cnt,cntA,cntB : integer;
     duplicate : boolean;
 begin (* procedure TempCircleMax *)
  cnt := 0;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do
   (* compute B(bx,by) * A(ax,ay) for template A centered    *)
   (* on B(bx,by) and maximize the product in C(ax+bx,ay+by) *)
   for cntA := 1 to A.num do begin
    duplicate := false
    (* find the pixel in the template configuration if it exists *)
    for i := 1 to cnt do
     if (C.cfg(.i.).r = A.cfg(.cntA.).r + B.cfg(.cntB.).r) and
        (C.cfg(.i.).c = A.cfg(.cntA.).c + B.cfg(.cntB.).c)
      then begin (* pixel already in template C configuration *)
       if A.cfg(.cntA.).w * B.cfg(.cntB.).w > C.cfg(.i.).w
        then C.cfg(.i.).w := A.cfg(.cntA.).w * B.cfg(.cntB.).w;
       duplicate := true;
       end; (* if pixel found *)
    if duplicate = false then begin (* append nonintersecting pixels *
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c
     C.cfg(.cnt.).w := A.cfg(.cntA.).w * B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cnt A do *)
   C.num := cnt;
   end; (* procedure TempCircleMax *)
```

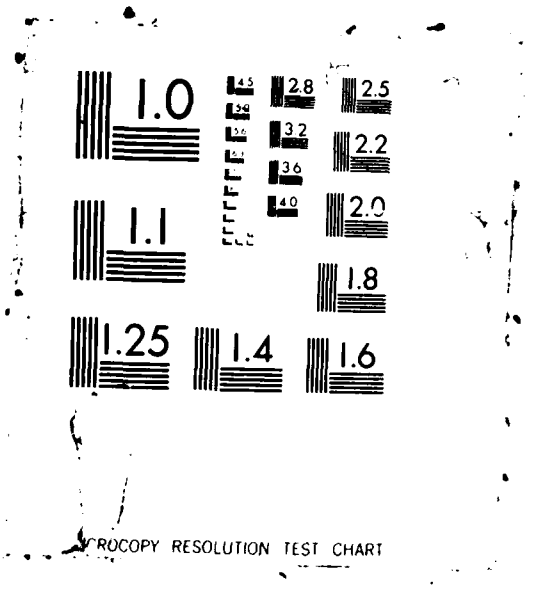
```
procedure TempCircleMin ( A,B : Template; var C : Template);
 var i,cnt,cntA,cntB : integer;
     duplicate : boolean;
 begin (* procedure TempCircleMin *)
  cnt := 0;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do
   (* compute B(bx,by) + A(ax,ay) for template A centered    *)
   (* on B(bx,by) and minimize the product in C(ax+bx,ay+by) *)
   for cntA := 1 to A.num do begin
    duplicate := false;
    (* find the pixel in the template configuration if it exists *)
    for i := 1 to cnt do
     if (C.cfg(.i.).r = A.cfg(.cntA.).r + B.cfg(.cntB.).r) and
        (C.cfg(.i.).c = A.cfg(.cntA.).c + B.cfg(.cntB.).c)
      then begin (* pixel already in template C configuration *)
       if A.cfg(.cntA.).w + B.cfg(.cntB.).w < C.cfg(.i.).w
        then C.cfg(.i.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
       duplicate := true;
       end; (* if pixel found *)
    if duplicate = false then begin (* append nonintersecting pixels *)
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cnt A do *)
  C.num := cnt
  end; (* procedure TempCircleMin *)

procedure TempSquareMax ( A,B : Template; var C : Template);
 var i,cnt,cntA,cntB : integer;
     duplicate : boolean;
 begin (* procedure TempSquareMax *)
   cnt := 0;
   (* for each pixel in template B *)
  for cntB := 1 to B.num do
    (* compute B(bx,by) + A(ax,ay) for template A centered *)
    (* on B(bx,by) and maximize the sum in C(ax+bx,ay+by) *)
   for cntA := 1 to A.num do begin
    duplicate := false;
    (* find the pixel in the template configuration if it exists *)
    for i := 1 to cnt do
     if (C.cfg(.i.).r = A.cfg(.cntA.).r + B.cfg(.cntB.).r) and
        (C.cfg(.i.).c = A.cfg(.cntA.).c + B.cfg(.cntB.).c)
      then begin (* pixel already in template C configuration *)
       if A.cfg(.cntA.).w + B.cfg(.cntB.).w > C.cfg(.i.).w
        then C.cfg(.i.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
       duplicate := true;
       end; (* if pixel found *)
    if duplicate = false then begin (* append nonintersecting pixels *)
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cntA do *)
  C.num := cnt
  end; (* procedure TempSquareMax *)
```

```
    if duplicate = false then begin (* append nonintersecting pixels *)
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cnt A do *)
  C.num := cnt;
  end; (* procedure TempSquareMax *)

procedure TempSquareMin ( A,B : Template; var C : Template);
 var i,cnt,cntA,cntB : integer;
     duplicate : boolean;
 begin (* procedure TempSquareMin *)
  cnt := 0;
  (* for each pixel in template B *)
  for cntB := 1 to B.num do
   (* compute B(bx,by) + A(ax,ay) for template A centered *)
   (* on B(bx,by) and minimize the sum in C(ax+bx,ay+by)  *)
   for cntA := 1 to A.num do begin
    duplicate := false;
    (* find the pixel in the template configuration if it exists *)
    for i := 1 to cnt do
     if (C.cfg(.i.).r = A.cfg(.cntA.).r + B.cfg(.cntB.).r) and
        (C.cfg(.i.).c = A.cfg(.cntA.).c + B.cfg(.cntB.).c)
      then begin (* pixel already in template C configuration *)
       if A.cfg(.cntA.).w + B.cfg(.cntB.).w < C.cfg(.i.).w
        then C.cfg(.i.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
       duplicate := true;
       end; (* if pixel found *)
    if duplicate = false then begin
     cnt := cnt + 1;
     C.cfg(.cnt.).r := A.cfg(.cntA.).r + B.cfg(.cntB.).r;
     C.cfg(.cnt.).c := A.cfg(.cntA.).c + B.cfg(.cntB.).c;
     C.cfg(.cnt.).w := A.cfg(.cntA.).w + B.cfg(.cntB.).w;
     end; (* if duplicate false *)
    end; (* for cnt A do *)
  C.num := cnt;
  end; (* procedure TempSquareMin *)

procedure TempScalarMultiply (A : Template; value : TemplateType; var C : Template);
 (* multiplication of an image by a scalar: C := A * value *)
 var cnt : integer;
 begin (* procedure TempScalarMultiply *)
  C := A;
  for cnt := 1 to A.num do C.cfg(.cnt.).w := C.cfg(.cnt.).w * value;
  end; (* procedure TempScalarMultiply *)
```

AD-A177 943 A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING C J TITUS DEC 86 AFIT/GE/ENG/86D-59 2/2

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
ROCOPY RESOLUTION TEST CHART

```
(*****************************************************************************)
(*                    BINARY IMAGE - TEMPLATE OPERATIONS                     *)
(*****************************************************************************)

procedure ImageTempCirclePlus ( A : Image; B : Template; var C : Image);
 (* convolve A with template B *)
 var Row,Col,i,j,cnt : integer;
     Sum : ImageType;
 begin (* procedure ImageTempCirclePlus *)
  C.row := A.row; C.col := A.col;
  for Row := 1 to A.row do
   for Col := 1 to A.col do begin
    (* sum the products of image pixels and overlayed template pixels *)
    Sum := 0;
    for cnt := 1 to B.num do begin
     i := B.cfg(.cnt.).r; j := B.cfg(.cnt.).c;
     if ((Row+i)>=1) and (Row+i<=A.row) and (Col+j)>=1) and (Col+j<=A.col))
      then Sum := Sum + A.gray(.Row+i,Col+j.) * B.cfg(.cnt.).w;
     end; (* for cnt do *)
    (* set the new pixel value equal to the sum *)
    C.gray(.Row,Col.) := Sum;
   end; (* for Col do *)
  end; (* procedure ImageTempCirclePlus *)

procedure ImageTempCircleMax ( A : Image; B : Template; var C : Image);
 var Row,Col,i,j,cnt : integer;
     MaxVal : ImageType;
 begin (* procedure ImageTempCircleMax *)
  C.row := A.row; C.col := A.col;
  ConstImage (C,A.row,A.col,0);
  (* for each pixel position *)
  for Row := 1 to A.row do
   for Col := 1 to A.col do begin
    (* set maxval to the product of present image pixel and template pixel center *)
    cnt := 1;
    while (B.cfg(.cnt.).r <> 0) and (B.cfg(.cnt.).c <> 0) do cnt := cnt + 1;
    if (B.cfg(.cnt.).r = 0) and (B.cfg(.cnt.).c = 0)
     then MaxVal := A.gray(.Row,Col.) * B.cfg(.cnt.).w (* template center defined *)
     else MaxVal := 0;                                 (* template center undefined *)
    (* search for the local maximum product *)
    for cnt := 1 to B.num do begin
     i := B.cfg(.cnt.).r; j := B.cfg(.cnt.).c;
     if ((Row+i)>=1) and (Row+i<=A.row) and (Col+j)>=1) and (Col+j<=A.col)) then
      if (A.gray(.Row+i,Col+j.) * B.cfg(.cnt.).w) > MaxVal
       then MaxVal := A.gray(.Row+i,Col+j.) * B.cfg(.cnt.).w;
     end; (* for cnt do *)
    (* set the new pixel value equal to the maximum *)
    C.gray(.Row,Col.) := MaxVal;
    end; (* for Col do *)
  end; (* procedure ImageTempCircleMax *)
```

```
procedure ImageTempCircleMin ( A : Image; B : Template; var C : Image);
 var Row,Col,i,j,cnt : integer;
     MinVal : ImageType;
 begin (* procedure ImageTempCircleMin *)
  C.row := A.row; C.col := A.col;
  ConstImage (C,A.row,A.col,0);
  (* for each pixel position *)
  for Row := 1 to A.row do
   for Col := 1 to A.col do begin
    (* set minval to the product of present image pixel and template center pixel *)
    cnt := 1;
    while (B.cfg(.cnt.).r <> 0) and (B.cfg(.cnt.).c <> 0) do cnt := cnt + 1;
    if (B.cfg(.cnt.).r = 0) and (B.cfg(.cnt.).c = 0)
     then MinVal := A.gray(.Row,Col.) * B.cfg(.cnt.).w (* template center defined *)
     else MinVal := 0;                                  (* template center undefined *)
    (* search for the local minimum product *)
    for cnt := 1 to B.num do begin
     i := B.cfg(.cnt.).r;
     j := B.cfg(.cnt.).c;
     if ((Row+i)>=1) and (Row+i<=A.row) and (Col+j)>=1) and (Col+j<=A.col)) then
      if (A.gray(.Row+i,Col+j.) * B.cfg(.cnt.).w) < MinVal
       then MinVal := A.gray(.Row+i,Col+j.) * B.cfg(.cnt.).w;
     end; (* for cnt do *)
    (* set the new pixel value equal to the minimum *)
    C.gray(.Row,Col.) := MinVal;
    end; (* for Col do *)
  end; (* procedure ImageTempCircleMin *)

procedure ImageTempSquareMax ( A : Image; B : Template; var C : Image);
 var Row,Col,i,j,cnt : integer;
     MaxVal : ImageType;
 begin (* procedure ImageTempSquareMax *)
  C.row := A.row; C.col := A.col;
  ConstImage (C,A.row,A.col,0);
  (* for each pixel position *)
  for Row := 1 to A.row do
   for Col := 1 to A.col do begin
    (* set maxval to the sum of present image pixel and template pixel center *)
    cnt := 1;
    while (B.cfg(.cnt.).r <> 0) and (B.cfg(.cnt.).c <> 0) do cnt := cnt + 1;
    if (B.cfg(.cnt.).r = 0) and (B.cfg(.cnt.).c = 0)
     then MaxVal := A.gray(.Row,Col.) + B.cfg(.cnt.).w (* template center defined *)
     else MaxVal := A.gray(.Row,Col.);                 (* template center undefined *)
    (* search for the local maximum sum *)
    for cnt := 1 to B.num do begin
     i := B.cfg(.cnt.).r; j := B.cfg(.cnt.).c;
     if ((Row+i)>=1) and (Row+i<=A.row) and (Col+j)>=1) and (Col+j<=A.col)) then
      if (A.gray(.Row+i,Col+j.) + B.cfg(.cnt.).w) > MaxVal
       then MaxVal := A.gray(.Row+i,Col+j.) + B.cfg(.cnt.).w;
     end; (* for cnt do *)
```

```
    (* set the new pixel value equal to the maximum *)
    C.gray(.Row,Col.) := MaxVal;
    end; (* for Col do *)
  end; (* procedure ImageTempSquareMax *)

procedure ImageTempSquareMin ( A : Image; B : Template; var C : Image);
 var Row,Col,i,j,cnt : integer;
     MinVal : ImageType;
 begin (* procedure ImageTempSquareMin *)
  C.row := A.row; C.col := A.col;
  ConstImage (C,A.row,A.col,0);
  (* for each pixel position *)
  for Row := 1 to A.row do
   for Col := 1 to A.col do begin
    (* set minval to the sum of present image pixel and template center pixel *)
    cnt := 1;
    while (B.cfg(.cnt.).r <> 0) and (B.cfg(.cnt.).c <> 0) do cnt := cnt + 1;
    if (B.cfg(.cnt.).r = 0) and (B.cfg(.cnt.).c = 0)
     then MinVal := A.gray(.Row,Col.) + B.cfg(.cnt.).w (* template center defined *)
     else MinVal := A.gray(.Row,Col.);                 (* template center undefined *)
    (* search for the local minimum sum *)
    for cnt := 1 to B.num do begin
     i := B.cfg(.cnt.).r;
     j := B.cfg(.cnt.).c;
     if ((Row+i>=1) and (Row+i<=A.row) and (Col+j>=1) and (Col+j<=A.col)) then
      if (A.gray(.Row+i,Col+j.) + B.cfg(.cnt.).w) < MinVal
       then MinVal := A.gray(.Row+i,Col+j.) + B.cfg(.cnt.).w;
     end; (* for cnt do *)
    (* set the new pixel value equal to the minimum *)
    C.gray(.Row,Col.) := MinVal;
    end; (* for Col do *)
  end; (* procedure ImageTempSquareMin *)
end. (* Module Real_IA_Operations *)
```

Appendix B: AFIT Image Algebra Preprocessor

```
program ImageAlgebraPreprocessor (input,output);

  (* This program reads a file containing the Image Algebra description of  *)
  (* an image processing algorithm, translates it to VAX PASCAL source code *)
  (* and procedure calls, and outputs the results to an external file for   *)
  (* subsequent compilation.  This program is written in VAX PASCAL.        *)
  (* Capt Christopher J Titus, Air Force Institute of Technology, GE-86D    *)

  (* global constants, types, and variables *)
  const MaxNode    =  50;     (* maximum number of nodes in the binary tree   *)
        BufferLen  = 136;     (* maximum character length of IA command line  *)
        NameLen    =  80;     (* maximum character length of variable names   *)
        MaxNames   = 100;     (* maximum number of variable names of one type *)
        DefaultImage    = 'real'; (* default data type of image gray levels *)
        DefaultTemplate = 'real'; (* default data type of template weights  *)
  type  BufferStr = varying[BufferLen] of char;
        NameStr   = varying[NameLen] of char;
        IntermedOper = record exist : boolean; end;
        Tree = array (.1..MaxNode.) of record (*  binary tree implemented with  *)
                   value  : NameStr;          (* a double linked list structure *)
                   parent : integer;          (* using an array for data storage *)
                   lchild : integer;
                   rchild : integer;
                   end; (* tree record *)
        VariableList = array (.1..MaxNames.) of varying[NameLen] of char;

  var   iR1,iR2,iR3,iR4,iR5,tR1,tR2,tR3,tR4,tR5 : IntermedOper;
        Name,Indent,TempName : NameStr;
        buffer,oldbuffer : BufferStr;
        ImageList,TemplateList,ScalarList : VariableList;
        NumImageNames,NumTemplateNames,NumScalarNames : integer;
        InFile,TmpFile,Pfile : text;
        Expression : Tree;
        i,j,nodeptr,nodecnt,code : integer;
        ImageType,Templatetype : varying[7] of char;
        DeclaredType : varying[4] of char;
```

```
procedure AssignIntermedResult (operitype : char; operator : NameStr;
                                var Result : NameStr);
 (* assign an intermediate result for the operation *)
 begin (* procedure AssignIntermedResult *)
  Result := ' ';
  if (operator <> ':=') and (operator <> '.') then begin
   (* assign an intermediate result for the operation *)
   if operitype = 'i' then begin
    (* operation is image-image, image-template, image-scalar, or unary  *)
    (* image; the result is an image; assign intermediate image operands *)
    if iR1.exist = false
     then begin Result := 'iR1'; iR1.exist := true; end
     else if iR2.exist = false
      then begin Result := 'iR2'; iR2.exist := true; end
      else if iR3.exist = false
       then begin Result := 'iR3'; iR3.exist := true; end
       else if iR4.exist = false
        then begin Result := 'iR4'; iR4.exist := true; end
        else if iR5.exist = false
         then begin Result := 'iR5'; iR5.exist := true; end
         else begin
          Result := '?';
          writeln;
          writeln ('ERROR no more intermediate images available ERROR');
          writeln ('ERROR   expression can not be evaluated     ERROR');
          end;
    end; (* if image operation *)
   if operitype = 't' then begin
    (* operation is template-template or template-scalar; the result *)
    (* is a template; release intermediate template operands *)
    if tR1.exist = false
     then begin Result := 'tR1'; tR1.exist := true; end
     else if tR2.exist = false
      then begin Result := 'tR2'; tR2.exist := true; end
      else if tR3.exist = false
       then begin Result := 'tR3'; tR3.exist := true; end
       else if tR4.exist = false
        then begin Result := 'tR4'; tR4.exist := true; end
        else if tR5.exist = true
         then begin Result := 'tR5'; tR5.exist := true; end
         else begin
          Result := '?';
          writeln;
          writeln ('ERROR no more intermediate templates available ERROR');
          writeln ('ERROR    expression can not be evaluated       ERROR');
          end;
    end; (* if template operation *)
   end; (* if operator then *)
  end; (* procedure AssignIntermedResult *)
```

```
procedure AssignResults (operand1,operand2,operator : NameStr;
                         var Result : NameStr;
                         var express:Tree; nodeptr:integer);
 (* update the expression tree to reflect the result   *)
 (* of the operation; release any intermediate results *)
 begin (* procedure AssignResults *)
  if operator = ':=' then Result := operand2;
  if operator = '.'  then Result := 'dotval';
  (* update the expression tree to reflect the operation result *)
  express(.nodeptr.).value := Result;
  (* release unused intermediate operators *)
  if ((operand1='iR1') or (operand2='iR1')) and (Result<>'iR1') then iR1.exist := false;
  if ((operand1='iR2') or (operand2='iR2')) and (Result<>'iR2') then iR2.exist := false;
  if ((operand1='iR3') or (operand2='iR3')) and (Result<>'iR3') then iR3.exist := false;
  if ((operand1='iR4') or (operand2='iR4')) and (Result<>'iR4') then iR4.exist := false;
  if ((operand1='iR5') or (operand2='iR5')) and (Result<>'iR5') then iR5.exist := false;
  if ((operand1='tR1') or (operand2='tR1')) and (Result<>'tR1') then tR1.exist := false;
  if ((operand1='tR2') or (operand2='tR2')) and (Result<>'tR2') then tR2.exist := false;
  if ((operand1='tR3') or (operand2='tR3')) and (Result<>'tR3') then tR3.exist := false;
  if ((operand1='tR4') or (operand2='tR4')) and (Result<>'tR4') then tR4.exist := false;
  if ((operand1='tR5') or (operand2='tR5')) and (Result<>'tR5') then tR5.exist := false;
  if (operand1='?') or (operand2='?') then express(.nodeptr.).value := '?';
  express(.nodeptr.).lchild := 0;
  express(.nodeptr.).rchild := 0;
  end; (* procedure AssignResults *)

procedure BinaryOperation (var express : Tree; nodeptr : integer);
 (* write the binary operation procedure calls to the output file *)
 var operator,operand1,operand2,Result,tempoper : NameStr;
     oper1type,oper2type,temptype : char;
 begin (* procedure BinaryOperation *)
  (* retrieve the operands from the binary tree *)
  operand1 := express(.express(.nodeptr.).lchild.).value;
  operator := express(.nodeptr.).value;
  operand2 := express(.express(.nodeptr.).rchild.).value;
  (* determine the operand types *)
  oper1type := ' '; oper2type := ' ';
  for i := 1 to NumImageNames do begin
   if operand1 = ImageList(.i.) then oper1type := 'i';
   if operand2 = ImageList(.i.) then oper2type := 'i';
   end; (* for i do *)
  for i := 1 to NumTemplateNames do begin
   if operand1 = TemplateList(.i.) then oper1type := 't';
   if operand2 = TemplateList(.i.) then oper2type := 't';
   end; (* for i do *)
  for i := 1 to NumScalarNames do begin
   if operand1 = ScalarList(.i.) then oper1type := 's';
   if operand2 = ScalarList(.i.) then oper2typ  := 's';
   end; (* for i do *)
```

```
if operitype = ' ' then begin
 operitype := 's';
 for i := 1 to length(operand1) do
  if (operitype = 's') and (operand1[i] in ['-','0'..'9'])
   then operitype := 's'
   else operitype := ' ';
 end; (* if operitype then *)
if oper2type = ' ' then begin
 oper2type := 's';
 for i := 1 to length(operand2) do
  if (oper2type = 's') and (operand2[i] in ['-','0'..'9'])
   then oper2type := 's'
   else oper2type := ' ';
 end; (* if oper2type then *)
if operitype = ' ' then
 writeln ('ERROR ',operand1,' is not an image, template or scalar operand  ERROR');
if oper2type = ' ' then
 writeln ('ERROR ',operand2,' is not an image, template or scalar operand  ERROR');

if (operand1 <> '?') and (operand2 <> '?') then begin
 write (Pfile,Indent);
 (* arrange the operands in the ;roper order: i-s,i-t,t-s *)
 if ((operitype = 's') and (oper2type = 'i')) or
    ((operitype = 't') and (oper2type = 'i')) or
    ((operitype = 's') and (oper2type = 't')) then begin
  tempoper := operand1;
  temptype := operitype;
  operand1 := operand2;
  operitype := oper2type;
  operand2 := tempoper;
  oper2type := temptype;
  end; (* arrange operands *)
 AssignIntermedResult (operitype,operator,Result);

 if (operitype = 'i') and (oper2type = 'i') then begin
  (* image-image binary operations *)
  if operator[1] in ['+','-','*','/','v','^',',','.',':','<','=','>']
   then case operator[1] of
    '+' : writeln (Pfile,'ImageAdd (',operand1,',',operand2,',',Result,');');
    '-' : writeln (Pfile,'ImageSubtract (',operand1,',',operand2,',',Result,');');
    '*' : if (length(operator) > 1) and (operator[2] = '*')
           then writeln (Pfile,'ImageExponent (',operand1,',',operand2,',',Result,');')
           else writeln (Pfile,'ImageMultiply (',operand1,
                              ',',operand2,',',Result,');');
    '/' : writeln (Pfile,'ImageDivide (',operand1,',',operand2,',',Result,');');
    'v' : writeln (Pfile,'ImageMax (',operand1,',',operand2,',',Result,');');
    '^' : writeln (Pfile,'ImageMin (',operand1,',',operand2,',',Result,');');
    '.' : writeln (Pfile,'ImageDot (',operand1,',',operand2,',dotval);');
    ':' : writeln (Pfile,operand2,' := ',operand1,';');
```

```
    otherwise writeln (PFile,'CharImage (',operand1,
                              ',''',operator,''',',operand2,',',Result,');');
    end (* case operator *)
   else begin
    writeln;
    writeln ('ERROR   operator ',operator,' does not exist   ERROR');
    writeln ('ERROR   for image-image operations    ERROR');
    operand1 := '?'; operand2 := '?';
    end;
  end; (* image-image binary operations *)

 if (oper1type = 'i') and (oper2type = 't') then begin
  (*image-template binary operations *)
  if (operator[1] in ['(','[']) and (operator[2] in ['+','v','^'])
   then begin
    if operator[1] = '(' then
     case operator[2] of
      '+' : writeln (Pfile,'ImageTempCirclePlus (',operand1,
                           ',',operand2,',',Result,');');
      'v' : writeln (Pfile,'ImageTempCircleMax (',operand1,
                           ',',operand2,',',Result,');');
      '^' : writeln (Pfile,'ImageTempCircleMin (',operand1,
                           ',',operand2,',',Result,');');
     end; (* case operator *)
    if operator[1] ='[' then
     case operator[2] of
      'v' : writeln (Pfile,'ImageTempSquareMax (',operand1,
                           ',',operand2,',',Result,');');
      '^' : writeln (Pfile,'ImageTempSquareMin (',operand1,
                           ',',operand2,',',Result,');');
     end; (* case operator *)
    end (* if operator then *)
   else begin
    writeln;
    writeln ('ERROR    operator ',operator,' does not exist      ERROR');
    writeln ('ERROR   for image-template operations   ERROR');
    operand1 := '?'; operand2 := '?';
    end;
  end; (* image-template binary operations *)
```

```
if (oper1type = 't') and (oper2type = 't') then begin
 (* template-template binary operations *)
 if (operator[1] in ['+','-','*','/','v','^',':']) or
    ((operator[1] in ['(','[']) and (operator[2] in ['+','v','^']))
  then begin
   if operator[1] = '(' then
    case operator[2] of
     '+' : writeln (Pfile,'TempCirclePlus (',operand1,',',operand2,',',Result,');');
     'v' : writeln (Pfile,'TempCircleMax (',operand1,',',operand2,',',Result,');');
     '^' : writeln (Pfile,'TempCircleMin (',operand1,',',operand2,',',Result,');');
     end; (* case operator *)
   if operator[1] = '[' then
    case operator[2] of
     'v' : writeln (Pfile,'TempSquareMax (',operand1,',',operand2,',',Result,');');
     '^' : writeln (Pfile,'TempSquareMin (',operand1,',',operand2,',',Result,');');
     end; (* case operator *)


   if (operator[1] in ['(','[']) = false then
    case operator[1] of
     '+' : writeln (Pfile,'TempAdd (',operand1,',',operand2,',',Result,');');
     '-' : writeln (Pfile,'TempSubtract (',operand1,',',operand2,',',Result,');');
     '*' : writeln (Pfile,'TempMultiply (',operand1,',',operand2,',',Result,');');
     '/' : writeln (Pfile,'TempDivide (',operand1,',',operand2,',',Result,');');
     'v' : writeln (Pfile,'TempMax (',operand1,',',operand2,',',Result,');');
     '^' : writeln (Pfile,'TempMin (',operand1,',',operand2,',',Result,');');
     ':' : writeln (Pfile,operand2,' := ',operand1,';');
     end; (* case operator *)
   end
  else begin
   writeln;
   writeln ('ERROR     operator ',operator,' does not exist        ERROR');
   writeln ('ERROR   for template-template operations   ERROR');
   operand1 := '?'; operand2 := '?';
   end;
 en ; (* template-template binary operations *)
```

```
if (oper1type = 'i') and (oper2type = 's') then begin
 (* image-scalar binary operations *)
 if operator[1] in ['*','/','<','=','>']
  then case operator[1] of
   '*' : if (length(operator) > 1) and (operator[2] = '*')
         then begin
          writeln (PFile,'ConstImage (',Result,',',operand1,'.row,',
                                       operand1,'.col,',operand2,');');
          write (Pfile,Indent);
          writeln (Pfile,'ImageExponent (',operand1,',',Result,',',Result,');');
          end
         else begin
          writeln (PFile,'ConstImage (',Result,',',operand1,'.row,',
                                       operand1,'.col,',operand2,');');
          write (Pfile,Indent);
          writeln (Pfile,'ImageMultiply (',operand1,',',Result,',',Result,');');
          end;
   '/' : begin
         writeln (PFile,'ConstImage (',Result,',',operand1,'.row,',
                                      operand1,'.col,',operand2,');');
         write (Pfile,Indent);
         writeln (Pfile,'ImageDivide (',operand1,',',Result,',',Result,');');
         end;
   otherwise begin
         writeln (PFile,'ConstImage (',Result,',',operand1,'.row,',
                                      operand1,'.col,',operand2,');');
         write (Pfile,Indent);
         writeln (PFile,'CharImage (',operand1,
                        ','''',operator,'''',',Result,',',Result,');');
         end; (* case else *)
   end (* case operator *)

  else begin
   writeln;
   writeln ('ERROR   operator ',operator,' does not exist   ERROR');
   writeln ('ERROR   for image-scalar operations    ERROR');
   operand1 := '?'; operand2 := '?';
   end;
 end; (* image-scalar binary operations *)
```

```
  if (oper1type = 't') and (oper2type = 's') then begin
   (* template-scalar binary operations *)
   if operator[1] in ['*']
    then case operator[1] of
     '*' : writeln (Pfile,'TempScalarMultiply (',operand1,',',operand2,',',Result,');');
     end (* case operator *)
    else begin
     writeln;
     writeln ('ERROR   operator ',operator,' does not exist   ERROR');
     writeln ('ERROR   for template-scalar operations    ERROR');
     operand1 := '?'; operand2 := '?';
     end;
   end; (* template-scalar binary operations *)

  if (oper1type = 's') and (oper2type = 's') then begin
   (* scalar-scalar binary operations *)
   if operator = ':='
    then writeln (Pfile,operand2,' := ',operand1,';')
    else begin
     writeln;
     writeln ('ERROR   operator ',operator,' does not exist   ERROR');
     writeln ('ERROR   for scalar-scalar operations   ERROR');
     operand1 := '?'; operand2 := '?';
     end;
   end; (* scalar-scalar binary operations *)
  end; (* if operands <> ? then *)

  AssignResults (operand1,operand2,operator,Result,express,nodeptr);
  end; (* procedure BinaryOperation *)

procedure UnaryOperation (var express : Tree; nodeptr : integer);
 (* write the unary operation procedure calls to the output file *)
 var operator,operand1,operand2,Result : NameStr;
     oper1type,oper2type : char;
 begin (* procedure UnaryOperation *)
  (* retrieve the operand and operator from the binary tree *)
  operator := express(.nodeptr.).value;
  if express(.nodeptr.).lchild <> 0
   then operand1 := express(.express(.nodeptr.).lchild.).value
   else operand1 := express(.express(.nodeptr.).rchild.).value;
  operand2 := ' ';
```

```
(* determine the operand type *)
oper1type := ' '; oper2type := ' ';
for i := 1 to NumImageNames do if operand1 = ImageList(.i.) then oper1type := 'i';
for i := 1 to NumTemplateNames do
 if operand1 = TemplateList(.i.) then oper1type := 't';
for i := 1 to NumScalarNames do if operand1 = ScalarList(.i.) then oper1type := 's';
if oper1type = ' ' then begin
 oper1type := 's';
 for i := 1 to length(operand1) do
  if (oper1type = 's') and (operand1[i] in ['-','0'..'9'])
   then oper1type := 's'
   else oper1type := ' ';
 end; (* if oper1type then *)
if oper1type = ' ' then
 writeln ('ERROR  ',operand1,' is not an image, template, or scalar operand  ERROR');

if operand1 <> '?' then begin
 write (Pfile,Indent);
 AssignIntermedResult (oper1type,operator,Result);
 case oper1type of
  'i' : begin (* unary image operations *)
        if operator = '|' then
         writeln (Pfile,'AbsValImage (',operand1,',',Result,');');
        if operator = '-' then begin
         writeln (PFile,'ConstImage (',Result,
                        ',',operand1,'.row,',operand1,'.col,-1);');
         write (Pfile,Indent);
         writeln (Pfile,'ImageMultiply (',operand1,',',Result,',',Result,');');
         end;
        end; (* case unary image operations *)
  't' : begin (* unary template operations *)
        if operator = '|' then
         writeln (Pfile,'AbsValTemp (',operand1,',',Result,');');
        if operator = '-' then
         writeln (Pfile,'TempScalarMultiply (',operand1,',-1,',Result,');');
        end; (* case unary template operations *)
  end; (* case operand type *)
 end; (* if operand1 <> ? then *)

AssignResults (operand1,operand2,operator,Result,express,nodeptr);
end; (* procedure UnaryOperation *)
```

```
procedure IAParser (buffer : BufferStr; var express : Tree;
                    var nodeptr : integer);
 (* parse the IA expressions and build a binary tree of operands and operators *)
 var i,j,next : integer;
     letter : char;
     assignoper,absvalflag,EndOfOperand : boolean;
 begin (* procedure IAParser *)
  absvalflag := false;
  (* if an assignment exists in the expression, evaluate the right side *)
  assignoper := false;
  if index(buffer,':=') <> 0
   then begin
    assignoper := true;
    IAParser(substr(buffer,index(buffer,':=')+2,length(buffer)-index(buffer,':=')-1),
                    express,nodeptr);
    end (* then *)
   else begin nodecnt := 1; nodeptr := 1; end;
  i := 1;
  if assignoper = false then begin (* translate the IA expression *)
   while i < length(buffer) do begin
    letter := buffer[i];
    next := i + 1;

    if (letter in ['a'..'u','w'..'z','0'..'9']) or
       ((letter = '-') and (buffer[next] in ['0'..'9']))
     then begin (* image, template, scalar, or variable operand *)
      nodecnt := nodecnt + 1;
      if express(.nodeptr.).value = ''
       then express(.nodeptr.).lchild := nodecnt
       else express(.nodeptr.).rchild := nodecnt;
      express(.nodecnt.).parent := nodeptr;
      j := next;
      EndOfOperand := false;
      while (j < length(buffer)) and (EndOfOperand = false) do
       if (buffer[j] in ['a'..'u','w'..'z','_','0'..'9']) or
          ((buffer[j]='.') and (buffer[j+1] in ['0'..'9']))
         then j := j + 1
         else EndOfOperand := true;
      express(.nodecnt.).value := substr(buffer,i,j-i);
      i := j;
      express(.nodecnt.).lchild := 0;
      express(.nodecnt.).rchild := 0;
      end; (* if image, template, scalar, or variable operand *)
```

```
if letter = '{' then begin (* start parenthesis *)
 nodecnt := nodecnt + 1;
 if express(.nodeptr.).value = ''
  then express(.nodeptr.).lchild := nodecnt
  else express(.nodeptr.).rchild := nodecnt;
 express(.nodecnt.).parent := nodeptr;
 nodeptr := nodecnt;
 i := i + 1;
 end; (* if start parenthesis *)

if letter = '}' then begin (* finish parenthesis *)
 if express(.nodeptr.).parent <> 0
  then nodeptr := express(.nodeptr.).parent
  else begin
   nodecnt := nodecnt + 1;
   express(.nodeptr.).parent := nodecnt;
   express(.nodecnt.).lchild := nodeptr;
   nodeptr := nodecnt;
   end;
 i := i + 1;
 end; (* if finish parenthesis *)

if (letter in ['(','[','+','*','/','.','v','^','!','=','<','>']) or
   ((letter = '-') and (buffer[next] in ['a'..'u','w'..'z','(']))
 then begin (* algebraic operator encountered *)
  if express(.nodeptr.).value <> '' then begin
   (* chained algebraic operator encountered - get a new node *)
   nodecnt := nodecnt + 1;
   if express(.nodeptr.).rchild <> 0
    then begin (* chained binary operator encountered *)
     if express(.nodeptr.).parent <> 0
      then begin (* chained operator in parentheses, alter pointers *)
       (* place the new node between the present and parent node *)
       (* link the new node to the present node's parent *)
       if express(.express(.nodeptr.).parent.).lchild = nodeptr
        then express(.express(.nodeptr.).parent.).lchild := nodecnt
        else express(.express(.nodeptr.).parent.).rchild := nodecnt;
       express(.nodecnt.).parent := express(.nodeptr.).parent;
       end; (* then alter pointers *)

     (* link the new node to the present node *)
     express(.nodeptr.).parent := nodecnt;
     express(.nodecnt.).lchild := nodeptr;
     end (* chained binary operator *)
    else begin (* chained unary operator encountered *)
     if express(.nodeptr.).value = ''
      then express(.nodeptr.).lchild := nodecnt
      else express(.nodeptr.).rchild := nodecnt;
     express(.nodecnt.).parent := nodeptr;
     end; (* chained unary operator *)
```

```
      (* move to the new node *)
      nodeptr := nodecnt;
      end; (* chained algebraic operator encountered *)

     (* place the operator into the empty node *)
     if express(.nodeptr.).value = '' then begin
      if (letter = '[') or (letter = '(')
       then begin
        express(.nodeptr.).value := substr(buffer,i,3);
        i := i + 3; (* skip the operator charaters *)
        end (* if letter then *)
       else begin
        if ((letter = '*') and (buffer[next] = '*')) or
           ((letter in ['>','<']) and (buffer[next] in ['=','>']))
         then begin
          express(.nodeptr.).value := substr(buffer,i,2);
          i := i + 2;
          end
         else begin
          if absvalflag = false
           then express(.nodeptr.).value := letter;
          i := i + 1; (* skip the operator character *)
          if (letter = '|') and (absvalflag = false)
           then absvalflag := true
           else absvalflag := false;
          end; (* if letter else *)
        end; (* if letter else *)
      end; (* place operator into the empty node *)
     end; (* if algebraic operator *)

  if letter = ' ' then i := i + 1;
   end; (* while i do *)
  (* return to the top node *)
  while express(.nodeptr.).parent <> 0 do
   nodeptr := express(.nodeptr.).parent;
  end; (* if assignoper false *)

 (* if assign operator exists, place it at the top of the tree *)
 if assignoper = true then begin
  nodecnt := nodecnt + 1;
  express(.nodeptr.).parent := nodecnt;
  express(.nodecnt.).lchild := nodeptr;
  express(.nodecnt.).value  := ':=';
  nodeptr := nodecnt;
  nodecnt := nodecnt + 1;
  express(.nodeptr.).rchild := nodecnt;
  express(.nodecnt.).parent := nodeptr;
  express(.nodecnt.).value  := substr(buffer,1,index(buffer,':=')-1);
  end; (* if assignoper then *)
 end; (* procedure IAParser *)
```

```
procedure IATranslator (var express : Tree; nodeptr : integer);
 (* translate the IA expressions by evaluating the binary tree *)
 begin (* IATranslator *)
  if express(.nodeptr.).lchild <> 0
   then IATranslator(express,express(.nodeptr.).lchild);
  if express(.nodeptr.).rchild <> 0
   then IATranslator(express,express(.nodeptr.).rchild);
  if express(.nodeptr.).value <> ''
   then begin (* execute the operator *)
    if ((express(.nodeptr.).lchild<>0) and (express(.nodeptr.).rchild=0)) or
       ((express(.nodeptr.).lchild=0) and (express(.nodeptr.).rchild<>0))
     then UnaryOperation (express,nodeptr);
    if (express(.nodeptr.).lchild <> 0) and (express(.nodeptr.).rchild <> 0)
     then BinaryOperation (express,nodeptr);
    end (* operator execution *)
   else if express(.nodeptr.).parent <> 0 then begin
    (* remove an unnecessary pair of parentheses *)
    if express(.express(.nodeptr.).parent.).lchild = nodeptr
     then (* link the parent node's left child to this node's... *)
      if express(.nodeptr.).lchild <> 0
       then begin (* ...left child *)
        express(.express(.nodeptr.).parent.).lchild := express(.nodeptr.).lchild;
        express(.express(.nodeptr.).lchild.).parent := express(.nodeptr.).parent;
        end
       else begin (* ...right child *)
        express(.express(.nodeptr.).parent.).lchild := express(.nodeptr.).rchild;
        express(.express(.nodeptr.).rchild.).parent := express(.nodeptr.).parent;
        end;
    if express(.express(.nodeptr.).parent.).rchild = nodeptr
     then (* link the parent node's right child to this node's... *)
      if express(.nodeptr.).lchild <> 0
       then begin (* ...left child *)
        express(.express(.nodeptr.).parent.).rchild := express(.nodeptr.).lchild;
        express(.express(.nodeptr.).lchild.).parent := express(.nodeptr.).parent;
        end
       else begin (* ...right child *)
        express(.express(.nodeptr.).parent.).rchild := express(.nodeptr.).rchild;
        express(.express(.nodeptr.).rchild.).parent := express(.nodeptr.).parent;
        end;
    end; (* remove parentheses *)
 end; (* IATranslator *)

begin (* main program Image_Algebra_Preprocessor *)
 reset (input);
 rewrite (output);
```

```
(* initialize variables and set intermediate operands to false *)
iR1.exist := false; iR2.exist := false; iR3.exist := false;
iR4.exist := false; iR5.exist := false;
tR1.exist := false; tR2.exist := false; tR3.exist := false;
tR4.exist := false; tR5.exist := false;
buffer := '';
(* initialize the lists of image, template, and scalar variables *)
ImageList(.1.) := 'iR1';      ImageList(.2.) := 'iR2';
ImageList(.3.) := 'iR3';      ImageList(.4.) := 'iR4';
ImageList(.5.) := 'iR5';
NumImageNames := 5;
for i := NumImageNames+1 to MaxNames do ImageList(.i.) := '';
TemplateList(.1.) := 'tR1';   TemplateList(.2.) := 'tR2';
TemplateList(.3.) := 'tR3';   TemplateList(.4.) := 'iR4';
TemplateList(.5.) := 'iR5';
NumTemplateNames := 5;
for i := NumTemplateNames+1 to MaxNames do TemplateList(.i.) := '';
ScalarList(.1.) := 'dotval'; ScalarList(.2.) := 'minval';
ScalarList(.3.) := 'maxval';
NumScalarNames := 3;
for i := NumScalarNames+1 to MaxNames do ScalarList(.i.) := '';

(* retrieve an existing Image Algebra file to be translated *)
open(InFile,'translat.ia',unknown);
reset (InFile) ;
(* change all letters in the file to lower case;      *)
(* remove any comments and blank lines from the file *)
open(TmpFile,'translat.tmp',unknown);
rewrite (TmpFile);
writeln ('   ...translating letters to lower case; removing comments ');
readln (InFile,buffer);
while (eof(InFile)=false) do begin
 (* remove comments from the line *)
 while index(buffer,'(*') <> 0 do begin
  buffer := ' ' + buffer + ' '; (* required for proper operation of substr *)
  if index(buffer,'*)') <> 0
   then buffer := substr(buffer,1,index(buffer,'(*')-1) +
    substr(buffer,index(buffer,'*)')+2,length(buffer)-index(buffer,'*)')-1)
   else buffer := substr(buffer,1,index(buffer,'(*')-1);
  end; (* while comment do *)
 (* remove trailing blanks *)
 i := length(buffer);
 while (i > 1) and (buffer[i] = ' ') do i := i - 1;
 buffer := substr(buffer,1,i);
 (* translate all letters to lower case *)
 for i := 1 to length(buffer) do
  if ord(buffer[i]) in [65..90] then buffer[i] := chr(ord(buffer[i])+32);
```

```
 (* print non-blank lines to TmpFile *)
 if i > 1 then writeln (TmpFile,buffer);
 readln (InFile,buffer);
 end; (* while not eof *)

(* write the last line *)
writeln (TmpFile,buffer);
close (TmpFile);
close (InFile);

(* build the PASCAL file from 'translat.tmp' *)
open(InFile,'translat.tmp',unknown);
reset (InFile);
open(Pfile,'translat.pas',unknown);
rewrite (Pfile);
(* find the program name if it exists *)
readln (InFile,buffer);
if (index(buffer,'const ')=0) and (index(buffer,'type ' )=0) and
   (index(buffer,'var '  )=0) and (index(buffer,'begin ')=0)
 then Name := buffer
 else Name := 'Unnamed';

(* determine the data type of the image and template operands *)
ImageType    := DefaultImage;
TemplateType := DefaultTemplate;
(* search the declaration section *)
while index(buffer,'begin') = 0 do begin
 readln (InFile,buffer);
 if index(buffer,'type ') <> 0 then
  (* find the data type declarations if they exist *)
  while ((index(buffer,'const ') = 0) and (index(buffer,'var '  ) = 0) and
        (index(buffer,'begin') = 0)) do begin
   (* determine the data type of the images and templates *)
   if index(buffer,'itype') <> 0 then begin
    if index(buffer,'integer') <> 0 then ImageType := 'integer';
    if index(buffer,'real'   ) <> 0 then ImageType := 'real';
    end; (* if itype then *)
   if index(buffer,'ttype') <> 0 then begin
    if index(buffer,'integer') <> 0 then TemplateType := 'integer';
    if index(buffer,'real'   ) <> 0 then TemplateType := 'real';
    end; (* if ttype then *)
   readln (InFile,buffer);
   end; (* while not 'const ','var ', or 'begin' *)
 end; (* while not 'begin' *)

(* include the proper image algebra and input/output routines *)
if (ImageType = 'integer') and (TemplateType = 'integer')
 then writeln (Pfile,'[inherit(''iiaoper.env'',iio.env'')]     ')
 else writeln (Pfile,'[inherit(''riaoper.env'',rio.env'')]     ');
writeln (Pfile,'program ',Name,' (input,output);               ');
```

```
writeln ('   ...processing user-defined types, constants, and variables');
(* find and append any user defined constants here *)
reset (InFile);
(* search the entire declaration section for constants *)
readln (InFile,buffer);
while index(buffer,'begin') = 0 do begin
 readln (InFile,buffer);
 if index(buffer,'const ') <> 0 then
  (* process each additional constant declaration line *)
  while ((index(buffer,'type ') = 0) and (index(buffer,'var '  ) = 0) and
        (index(buffer,'begin') = 0)) do begin
   (* strip the reserved word 'type' from the buffer *)
   if index(buffer,'const ') <> 0
    then buffer := substr(buffer,index(buffer,'const ')+5,
                          length(buffer)-index(buffer,'const ')-4);
   (* remove all blanks from the line *)
   oldbuffer := buffer;
   buffer := '';
   for i := 1 to length(oldbuffer) do
    if oldbuffer[i] <> ' ' then buffer := buffer + oldbuffer[i];
   (* write the const declaration as is and ensure each line ends with a ';' *)
   if index(buffer,';') = 0 then writeln (Pfile,' const ',buffer,';')
                            else writeln (Pfile,' const ',buffer);
  readln (InFile,buffer);
  end; (* while not 'type','var', or 'begin', do *)
 end; (* while not 'begin' do *)

(* find and append any user defined types here *)
reset (InFile);
(* search the entire declaration section for types *)
readln (InFile,buffer);
while index(buffer,'begin') = 0 do begin
 readln (InFile,buffer);
 if index(buffer,'type ') <> 0 then
  (* process each additional type declaration line *)
  while ((index(buffer,'const ') = 0) and (index(buffer,'var '  ) = 0) and
        (index(buffer,'begin') = 0)) do begin
   (* write the type declaration as is if it is not itype or ttype *)
   if (index(buffer,'itype') = 0) and (index(buffer,'ttype') = 0) then begin
    (* strip the reserved word 'type' from the buffer *)
    if index(buffer,'type ') <> 0 then
     buffer := substr(buffer,index(buffer,'type ')+4,
                      length(buffer)-index(buffer,'type ')-3);
    (* remove all blanks from the line *)
    oldbuffer := buffer;
    buffer := '';
    for i := 1 to length(oldbuffer) do
     if oldbuffer[i] <> ' ' then buffer := buffer + oldbuffer[i];
    if index(buffer,';') = 0 then writeln (Pfile,' type ',buffer,';')
                             else writeln (Pfile,' type ',buffer);
```

```
    end; (* if type declaration *)
   readln (InFile,buffer);
   end; (* while not 'const ', 'var ', or 'begin' do *)
 end; (* while not 'begin' *)

(* find and append any user defined variables here *)
reset (InFile);
(* search the entire declaration section for variables *)
readln (InFile,buffer);
while index(buffer,'begin')=0 do begin
 readln (InFile,buffer);
 if index(buffer,'var ') <> 0 then
  (* process each additional constant declaration line *)
  while ((index(buffer,'const ') = 0) and (index(buffer,'type ') = 0) and
        (index(buffer,'begin') = 0)) do begin
   (* strip the reserved word 'var' from the buffer *)
   if index(buffer,'var ') <> 0
    then buffer := substr(buffer,index(buffer,'var ')+3,
                          length(buffer)-index(buffer,'var ')-2);
   (* remove all blanks from the line *)
   oldbuffer := buffer;
   buffer := '';
   for i := 1 to length(oldbuffer) do
    if oldbuffer[i] <> ' ' then buffer := buffer + oldbuffer[i];
   (* write the variable declaration as is and ensure each line ends with a ';' *)
   if index(buffer,';') = 0 then writeln (Pfile,' var ',buffer,';')
                            else writeln (Pfile,' var ',buffer);
   (* if the line contains any image, template, or scalar declarations, *)
   (* append these names to the appropriate name list    *)
   if (index(buffer,':image'   ) <> 0) or (index(buffer,' image'   ) <> 0) or
      (index(buffer,':template') <> 0) or (index(buffer,' template') <> 0) or
      (index(buffer,':integer' ) <> 0) or (index(buffer,' integer' ) <> 0) or
      (index(buffer,':real'    ) <> 0) or (index(buffer,' real'    ) <> 0) then begin
    (* determine the type of variable declarations *)
    DeclaredType := substr(buffer,index(buffer,':')+1,4);
    (* extract the variable names from the buffer *)
    i := 1;
    while i < index(buffer,':') do begin
     j := i;
     while (buffer[j] <> ':') and (buffer[j] <> ',') do j := j + 1;
     if (j-i) > 0 then begin
      if DeclaredType = 'imag' then begin
       NumImageNames := NumImageNames + 1;
       ImageList(.NumImageNames.) := substr(buffer,i,j-i);
       end; (* if image declarations then *)
      if DeclaredType = 'temp' then begin
       NumTemplateNames := NumTemplateNames + 1;
       TemplateList(.NumTemplateNames.) := substr(buffer,i,j-i);
       end; (* if template declarations then *)
```

```
      if (DeclaredType = 'inte') or (DeclaredType = 'real') then begin
       NumScalarNames := NumScalarNames + 1;
       ScalarList(.NumScalarNames.) := substr(buffer,i,j-i);
       end; (* if scalar declarations then *)
      end; (* if j-i then *)
     i := j + 1;
     end; (* while buffer[i] do *)
    end; (* if image, template, or scalar declaration *)
   readln (InFile,buffer);
   end; (* while not 'const','type', or 'begin', do *)
 end; (* while not 'begin' do *)

(* translate the IA file to PASCAL procedure calls *)
(* find the beginning of the executable code section *)
while index(buffer,'begin')=0 do readln (InFile,buffer);
writeln (PFile,' begin (* program ',Name,' *)                          ');
writeln (Pfile,'  reset (input);                                       ');
writeln (Pfile,'  rewrite (output);                                    ');

(* find and reposition any invariant template definitions here *)
while eof(InFile)=false do begin
 while ((index(buffer,'invariant')=0) or (index(buffer,'template')=0)) and
       (eof(InFile)=false) do readln (InFile,buffer);
 if (index(buffer,'invariant') <> 0) and (index(buffer,'template') <> 0) then begin
  (* extract the indentation from the declaration line *)
  Indent := '';
  i := 1;
  while buffer[i] = ' ' do begin Indent := Indent + ' '; i := i + 1; end;
  (* extract the template name from the declaration line *)
  i := index(buffer,'template') + 8;
  while buffer[i] = ' ' do i := i + 1;
  j := i;
  while (j < length(buffer)) and ( buffer[j] <> ' ') do j := j + 1;
  TempName := substr(buffer,i,j-i+1);
  (* read past the begin statement *)
  readln (InFile,buffer);
  (* read the first line of the template definition *)
  readln (InFile,buffer);
  j := 1;
  (* process each additional line of the template definition *)
  while index(buffer,'end') = 0 do begin
   (* write the necessary declarations for each template cell *)
   write (Pfile,Indent,TempName,'.cfg(.',j,'.).r:=');
   if (index(buffer,',c')-index(buffer,'(r')-1) > 2
    then write (Pfile,substr(buffer,index(buffer,'(r')+2,
                     index(buffer,',c')-index(buffer,'(r')-2),';')
    else write (Pfile,' 0;');
   write (Pfile,' ',TempName,'.cfg(.',j,'.).c:=');
```

```
   if (index(buffer,')=')-index(buffer,',c')) > 2
    then write (Pfile,substr(buffer,index(buffer,',c')+2,
                      index(buffer,')=')-index(buffer,',c')-2),';')
    else write (Pfile,' 0;');
   write (Pfile,' ',TempName,'.cfg(.',j,'.).w:=');

   if index(buffer,';') = 0
    then writeln (Pfile,substr(buffer,index(buffer,'=')+1,
                               length(buffer)-index(buffer,'=')),';')
    else writeln (Pfile,substr(buffer,index(buffer,'=')+1,
                               index(buffer,';')-index(buffer,'=')-1));
   j := j + 1;
   readln (Infile,buffer);
   end; (* while 'end' do *)
  writeln (Pfile,Indent,TempName,'.num := ',j-1,';');
  end; (* then invariant template definition *)
 end; (* while not eof *)

(* find the beginning of the executable portion of the routine *)
reset (InFile);
readln (InFile,buffer);
while index(buffer,'begin') = 0 do readln (Infile,buffer);

(* read each line of the file and translate accordingly *)
writeln ('   ...processing the executable statements');
readln (InFile,buffer);
while index(buffer,'end.') = 0 do begin
 (* if the line doesn't contain any PASCAL contructs or explicit *)
 (* IA procedures, then translate the IA expression to PASCAL    *)
 if ((index(buffer,'and '         ) = 0) and
     (index(buffer,'array'        ) = 0) and
     (index(buffer,'begin'        ) = 0) and
     (index(buffer,'case '        ) = 0) and
     (index(buffer,'const '       ) = 0) and
     (index(buffer,'do '          ) = 0) and
     (index(buffer,'downto '      ) = 0) and
     (index(buffer,'else '        ) = 0) and
     (index(buffer,'end'          ) = 0) and
     (index(buffer,'for '         ) = 0) and
     (index(buffer,'function '    ) = 0) and
     (index(buffer,'if '          ) = 0) and
     (index(buffer,'input'        ) = 0) and
     (index(buffer,'mod '         ) = 0) and
     (index(buffer,'not '         ) = 0) and
     (index(buffer,'of '          ) = 0) and
     (index(buffer,'or '          ) = 0) and
     (index(buffer,'output'       ) = 0) and
     (index(buffer,'procedure '   ) = 0) and
     (index(buffer,'program '     ) = 0) and
     (index(buffer,'record '      ) = 0) and
```

```
    (index(buffer,'string'      ) = 0) and
    (index(buffer,'then '       ) = 0) and
    (index(buffer,'type '       ) = 0) and
    (index(buffer,'to '         ) = 0) and
    (index(buffer,'until '      ) = 0) and
    (index(buffer,'var '        ) = 0) and
    (index(buffer,'while '      ) = 0) and
    (index(buffer,'with '       ) = 0) and
    (index(buffer,'boolean'     ) = 0) and
    (index(buffer,'byte'        ) = 0) and
    (index(buffer,'char'        ) = 0) and
    (index(buffer,'close'       ) = 0) and
    (index(buffer,'integer'     ) = 0) and
    (index(buffer,'open '       ) = 0) and
    (index(buffer,'read'        ) = 0) and
    (index(buffer,'real'        ) = 0) and
    (index(buffer,'reset'       ) = 0) and
    (index(buffer,'rewrite'     ) = 0) and
    (index(buffer,'text'        ) = 0) and
    (index(buffer,'true'        ) = 0) and
    (index(buffer,'write'       ) = 0) and
    (index(buffer,'image'       ) = 0) and
    (index(buffer,'configtemp'  ) = 0) and
    (index(buffer,'template'    ) = 0))
then begin (* translate the line from IA to PASCAL *)
 (* initialize the indentation for the line *)
 indent := '';
 i := 1;
 while buffer[i] = ' ' do begin
  indent := indent + ' ';
  i := i + 1;
  end;
 (* remove all blanks from the buffer *)
 oldbuffer := buffer;
 buffer := '';
 for i := 1 to length(oldbuffer) do
  if oldbuffer[i] <> ' ' then buffer := buffer + oldbuffer[i];
 (* delete the semi-colon from the end of the line *)
 if  index(buffer,';') <> 0 then buffer := substr(buffer,1,index(buffer,';')-1);
 (* translate parentheses to non-comment delimiters for writing to Pfile *)
 oldbuffer := buffer;
 for i := 1 to length(oldbuffer) do begin
  if oldbuffer[i] = '(' then oldbuffer[i] := '(';
  if oldbuffer[i] = ')' then oldbuffer[i] := ')';
  end; (* for i do *)
 (* print the IA expression to PFile *)
 writeln (Pfile,Indent,'(*****',oldbuffer,'*****)');
 (* pad the string with one blank on the end *)
 buffer:= buffer + ' ';
```

```
    (* initialize the expression binary tree to null *)
    for i := 1 to MaxNode do begin
     Expression(.i.).value := '';
     Expression(.i.).parent := 0;
     Expression(.i.).lchild := 0;
     Expression(.i.).rchild := 0;
     end;

    (* evaluate the expression and translate the line from IA to PASCAL *)
    IAParser (buffer,Expression,nodeptr);
    IATranslator (Expression,nodeptr);
    end (* translation from IA to PASCAL *)
   else begin (* PASCAL/IA construct or template definition *)
    (* translate parentheses to non-comment delimiters *)
    for i := 1 to length(buffer) do begin
     if buffer[i] = '(' then buffer[i] := '(';
     if buffer[i] = ')' then buffer[i] := ')';
     end; (* for i do *)
    if index(buffer,'variant') = 0
     then begin (* PASCAL/IA construct: write the line as is *)
      write (Pfile,buffer);
      (* ensure the lines end with a ';' unless the buffer ends   *)
      (* with the reserved words 'begin', 'do', 'then', or 'else' *)
      if (index(buffer,';')=0) and
         (index(buffer,'begin')<>length(buffer)-4) and
         (index(buffer,'do')<>length(buffer)-1) and
         (index(buffer,'then')<>length(buffer)-3) and
         (index(buffer,'else')<>length(buffer)-3)
        then writeln (Pfile,';')
        else writeln (Pfile);
       end (* PASCAL/IA construct *)
     else begin (* template definition encountered - skip it *)
      readln (InFile,buffer);
      while index(buffer,'end') = 0 do readln (InFile,buffer);
      end; (* else template definition *)
    end; (* else PASCAL/IA construct or template definition *)
  readln (InFile,buffer);
  end; (* while not EOF *)
writeln (Pfile,'  end. (* program ',Name,' *)');
close (Pfile);
end. (* procedure Image_Algebra_Preprocessor *)
```

Appendix C: AFIT Input / Output Operations

```
[inherit('riaoper.env'), environment('rio.env')]
Module Real_IA_IO_Operations (input,output);

procedure GetImage (var A : Image; FileName : NameStr);
  (* prompt and retrieve the requested image from disk *)
  var i,j : integer;
      row,col : real;
      InFile : file of real;
  begin (* procedure GetImage *)
   if FileName = '' then begin
    write ('read image from file: ');
    readln (FileName);
    end; (* if FileName *)
   if index(FileName,'.') = 0
    then FileName := FileName + '.img';
   open(InFile,FileName,unknown);
   reset (InFile) ;
   (* read in the input array *)
   read (InFile,row,col);
   A.row := round(row);
   A.col := round(col);
   for i := 1 to A.row do
    for j := 1 to A.col do read (InFile,A.gray(.i,j.));
   close (InFile);
   end; (* procedure GetImage *)

 procedure PutImage (A : Image; FileName : NameStr);
  (* write the image to an external file *)
  var i,j : integer;
      OutFile : file of real;
  begin (* procedure PutImage *)
   if index(FileName,'.') = 0
    then FileName := FileName + '.img';
   open(OutFile,FileName,unknown);
   rewrite (OutFile);
   write (OutFile,A.row,A.col);
   for i := 1 to A.row do
    for j := 1 to A.col do write (OutFile,A.gray(.i,j.));
   close (OutFile);
   end; (* procedure PutImage *)
```

```
procedure GetTemplate (var A : Template; FileName : NameStr);
 (* prompt and retrieve the requested template from disk *)
 var cnt : integer;
     rowoff,coloff,weight : real;
     InFile : file of real;
 begin (* procedure GetTemplate *)
  if FileName = '' then begin
   write ('read template from file: ');
   readln (FileName);
   end; (* if FileName *)
  if index(FileName,'.') = 0
   then FileName := FileName + '.tmp';
  open(InFile,FileName,unknown);
  reset (InFile) ;
  (* read in the input array *)
  cnt := 0;
  read (InFile,rowoff,coloff,weight);
  while (EOF(InFile)=false) do begin
   cnt := cnt + 1;
   A.cfg(.cnt.).r := round(rowoff);
   A.cfg(.cnt.).c := round(coloff);
   A.cfg(.cnt.).w := weight;
   read (InFile,rowoff,coloff,weight);
   end; (* while not EOF *)
  A.num := cnt;
  close (InFile);
  end; (* procedure GetTemplate *)

procedure PutTemplate (A : Template; FileName : NameStr);
 (* write the template to an external file *)
 var i : integer;
     OutFile : file of real;
 begin (* procedure PutTemplate *)
  if index(FileName,'.') = 0
   then FileName := FileName + '.tmp';
  open(OutFile,FileName,unknown);
  rewrite (OutFile);
  for i := 1 to A.num do
   write (OutFile,A.cfg(.i.).r,A.cfg(.i.).c,A.cfg(.i.).w);
  close (OutFile);
  end; (* procedure PutTemplate *)
end. (* Module Real_IA_IO_Operations *)
```

Appendix D: Automated Image Algebra Translator

```
$!  Translate Image Algebra
$!
$!  function - This command procedure translates an image
$!             processing function from its description in
$!             the image algebra to an executable file.
$!
$!  format   - @TIA [filename]
$!
$   on error then exit
$
$  if p1 .nes. "" then goto preprocess
$  inquire p1 "image algebra routine to be translated
$
$ preprocess:
$  copy 'p1'.ia translat.ia
$  write sys$output "...preprocessing"
$  r preproc
$
$  write sys$output "...compiling"
$  pas translat.pas
$
$  write sys$output "...linking"
$  link translat,iaoper,io
$
$  copy translat.exe 'p1'.exe
$  write sys$output "...executable file built"
$
$  del translat.*;*
$  purge
```

Bibliography

1. Ritter, Gerhard X. and others. Image Algebra Tutorial, Version 1, September 1985. Contract F08365-84-C-0295. University of Florida, Gainesville FL, September 1985.

2. Ritter, Gerhard X. and others. Image Algebra: Final Report for Phase I of the Project, September 1984 - September 1985. Contract F08365-84-C-0295. University of Florida, Gainesville FL, September 1985.

3. TurboPascal Version 3.0 Reference Manual. Borland International, Inc., Scotts Valley CA, 1985.

4. AA-L369B-TE. Programming in VAX Pascal. Digital Equipment Corporation, Maynard MA. March 1985.

5. Ritter, Gerhard X. and others. Image Algebra Project: Phase II, Program Review 2. Contract F088365-84-C-0295. University of Florida, Gainesville FL, March 1986.

6. Pratt, William K. Digital Image Processing. New York: John Wiley and Sons, Inc., 1978.

7. AA-D034D-TE. Programming in VAX Fortran. Digital Equipment Corporation, Maynard MA. September 1984.

8. Hancock, Les and Morris Krieger. The C Primer. New York: McGraw-Hill Book Company, 1982.

9. Conway, Richard and David Gries. An Introduction to Programming: A Structured Approach Using PL/I and PL/C. (Third Edition). Cambridge, Massachusetts: Winthrop Publishers, Inc., 1979.

VITA

Captain Christopher J. Titus was born on 3 December 1959 in Melrose, Massachusetts. He was graduated from Kingswood Regional High School in Wolfeboro, New Hampshire, in 1978 and attended Cornell University from which he received the degree of Bachelor of Science in Electrical Engineering in 1982. Upon graduation, he received a commission in the USAF through the ROTC program. He was employed as an analyst for the Foreign Technology Division at Wright-Patterson AFB, Ohio, until entering the School of Engineering, Air Force Institute of Technology, in May 1985.

permanent address: Keewaydin Rd, Star Route 1
Wolfeboro, New Hampshire 03894

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A177 995

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/86D-59	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Armament Laboratory 8b. OFFICE SYMBOL (If applicable) AFATL/ASE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Advanced Seeker Division Air Force Armament Laboratory Eglin AFB, Florida 32542	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A PASCAL IMPLEMENTATION OF THE IMAGE ALGEBRA	Approved for public release: IAW AFR 190-1. LYNN E. WOLAVER 5 March 87 Dean for Research and Professional Development Air Force Institute of Technology Wright-Patterson AFB OH 45433
12. PERSONAL AUTHOR(S) Titus, Christopher J., B.S., Captain, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____ 14. DATE OF REPORT (Year, Month, Day) 1986 December 15. PAGE COUNT 124	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	01	
17	07	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
image processing techniques; image algebra

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This study produced an image processing algorithm development tool on a VAX computer system using the recent advances in an Image Algebra developed by C. Ritter et al at the University of Florida. The image algebra provides the basis for a hardware and software independent environment for the expression of practically all image processing algorithms.

The goals of this project were twofold. The first goal was the implementation of the image algebra operators in a high level language to achieve hardware independence. The second goal was the design and implementation of a flexible preprocessor that could translate image processing algorithms, written in the image algebra language, into a high level computer language which could be compiled and executed on the VAX computer.

The implementation was achieved in the PASCAL computer language. All of the basic image algebra operators and the preprocessor were successfully programmed on the VAX

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr Matthew Kabrisky	22b TELEPHONE (Include Area Code) 513-255-3030 22c OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. computer, but a complete software independence of the image algebra was not achieved. This version also produces very large blocks of executable code for relatively simple algorithms.

Examples of the power and simplicity of the image algebra language and preprocessor environment are included.

END

4-87

DTIC